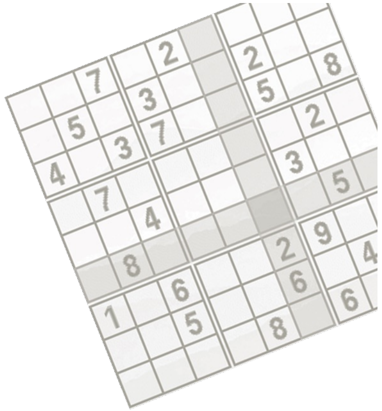
ESSAI & ACAI 2024
ATHENS - GREECE



Introduction to Constraint Satisfaction

Roman Barták
Charles University, Prague (CZ)

So far we have two methods to solve CSPs:

- **search**
  - complete (finds a solution or proves its non-existence)
  - too slow (exponential)
    - explores "visibly" wrong variable instantiations
- **consistency techniques**
  - usually incomplete (inconsistent values stay in domains)
  - pretty fast (polynomial)

Share advantages of both approaches - **combine** them!

- label the variables step by step (backtracking)
- maintain consistency after assigning a value

Do not forget about **traditional solving techniques**!

- linear equality solvers, simplex …
- such techniques can be integrated to global constraints!

Golomb&Baumert (1965), Bitner&Reingold (1975)

Backtracking

A core constraint satisfaction method:

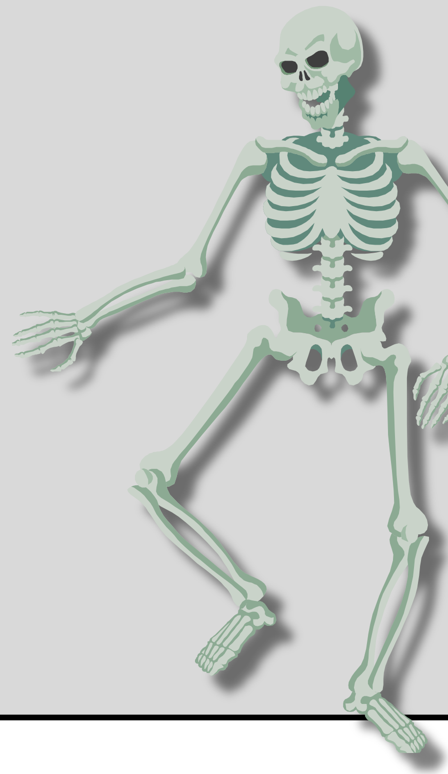
- **label (instantiate) the variables** one by one
  - the variables are ordered and instantiated in that order
- **verify (maintain) consistency** after each assignment

Skeleton of the search algorithm

```
procedure Labelling(G)
    return LBL(G,1)
end Labelling

procedure LBL(G,cv)
    if cv>|nodes(G)| then return nodes(G)
    for each value V from D_cv do
        if consistent(G,cv) then
            R ← LBL(G,cv+1)
            if R ≠ fail then return R
        end if
    end for
    return fail
end LBL
```

A „hook“ for consistency procedure



Look-back techniques

"Maintain" **consistency among the already instantiated variables**.

- „look back" = look to already labelled variables

What's result of consistency maintenance among labelled variables?

- a **conflict** (and/or its source - a violated constraint)

Backtracking is a basic look-back method.

Backward consistency checks

procedure AC-BT(G,cv)
    Q ← {(V_i,V_{cv}) in arcs(G), i<cv} % arcs to labelled variables.
    consistent ← true
    while consistent & Q non empty **do**
        select and delete any arc (V_k,V_m) from Q
        consistent ← not REVISE(V_k,V_m)
    end while
    return consistent
end AC-BT

When a value is deleted, the domain is empty

Backjumping & comp. uses information about the violated constraints.

McGregor (1975),Van Hentenryck (1989)

Forward checking

It is better to prevent failures than to detect them only!

Consistency techniques can remove incompatible values for future (=not yet instantiated) variables.

Forward checking ensures consistency between the currently instantiated variable and the variables connected to it via constraints.

Forward consistency checks

procedure AC-FC(G,cv)
    Q $\leftarrow$ {(V_i,V_{cv}) in arcs(G), i>cv}   % arcs to future variables
    consistent $\leftarrow$ true
    while consistent & Q non empty **do**
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) **then**
            consistent $\leftarrow$ not empty D_k
        end if
    end while
    return consistent
end AC-FC

Empty domain implies inconsistency

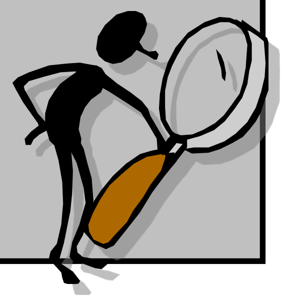
Partial look-ahead

We can extend the consistency checks to more future variables!

The value assigned to the current variable can be propagated to all future variables.

Partial look-ahead consistency checks

```
procedure DAC-LA(G,cv)
    for i=cv+1 to n do
        for each arc (Vi,Vj) in arcs(G) such that i>j & j≥cv do
            if REVISE(Vi,Vj) then
                if empty Di then return fail
        end for
    end for
    return true
end DAC-LA
```



Notes:

In fact DAC is maintained (in the order reverse to the labelling order).

Partial Look Ahead or **DAC - Look Ahead**

It is not necessary to check consistency of arcs between the future variables and the past variables (different from the current variable)!

Gaschnig (1974), Sabin&Freuder (1994)

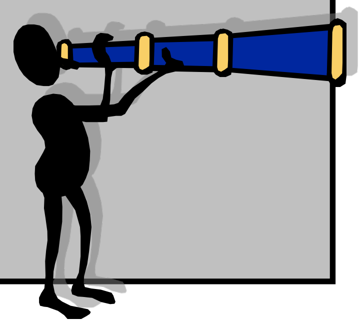
Full look-ahead

Knowing more about far future is an advantage!

Instead of DAC we can use a full AC (e.g. AC-3).

Full look ahead consistency checks

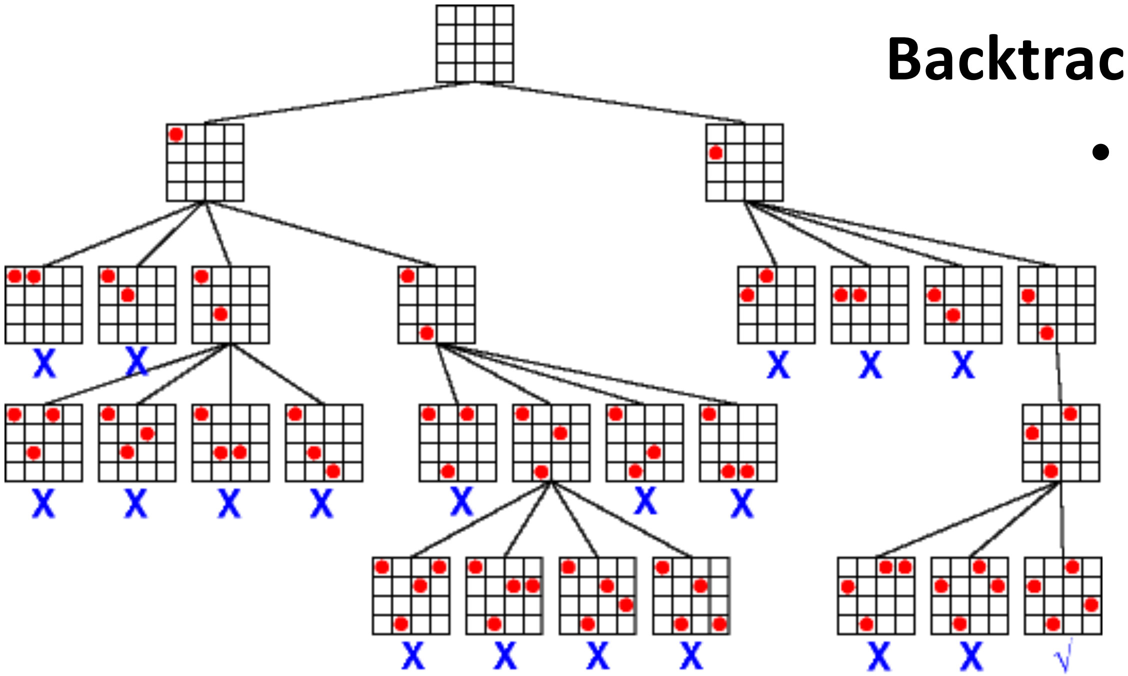
procedure AC3-LA(G,cv)
    $Q \leftarrow \{(V_i,V_{cv})$ in arcs(G),i>cv$\}$ % start with arcs going to cv
    consistent $\leftarrow$ true
    while consistent & Q non empty **do**
        select and delete any arc (V_k,V_m) from Q
        if REVISE(V_k,V_m) **then**
            $Q \leftarrow Q \cup \{(V_i,V_k) \mid (V_i,V_k)$ in arcs(G),$i\neq k, i\neq m, i>cv\}$
            consistent $\leftarrow$ not empty D_k
        end if
    end while
    return consistent
end AC3-LA



Notes:

- The arcs going to the current variable are checked exactly once.
- The arcs to past variables are not checked at all.
- It is possible to use other than AC-3 algorithms (e.g. AC-4)

Comparison of solving methods (4 queens)

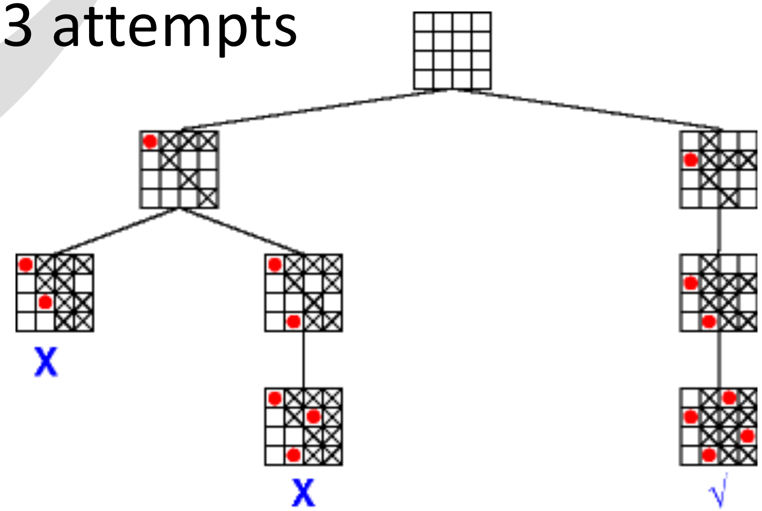


Backtracking is not very good

- 19 attempts

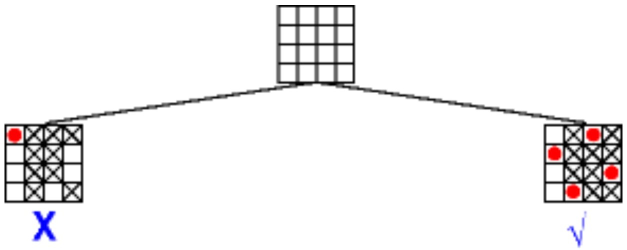
Forward checking is better

3 attempts

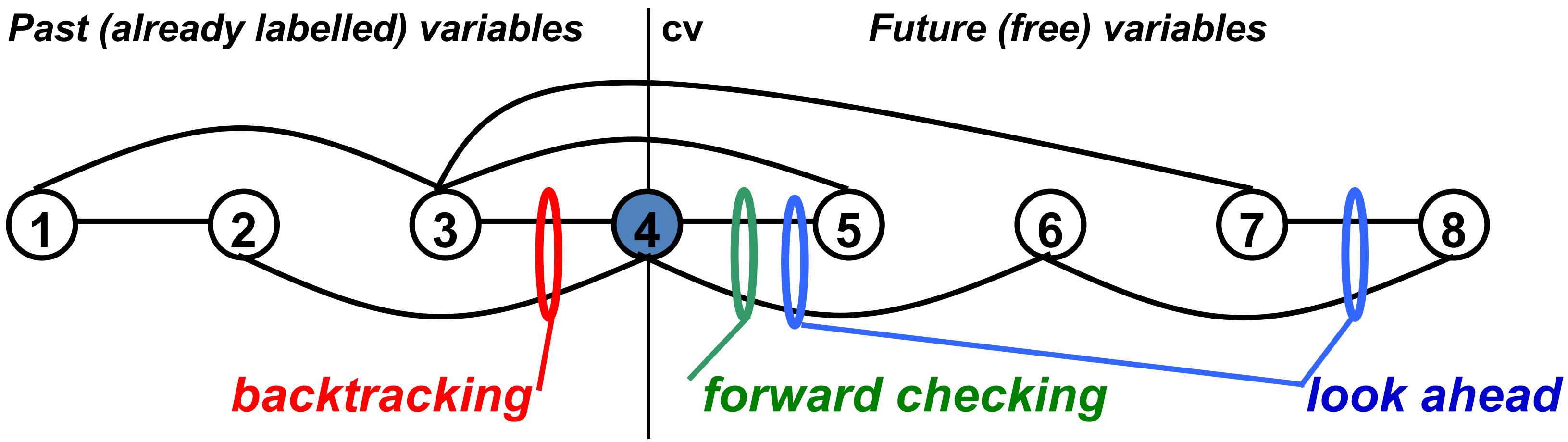


And the winner is **Look Ahead**

2 attempts



Constraint propagation at glance



- Propagating through more constraints removes more inconsistencies (BT < FC < PLA < LA), of course it increases complexity of the labelling step.
- **Forward Checking does no increase complexity of backtracking**, the constraint is just checked earlier in FC (BT tests it later).
- When using AC-4 in LA, the initialisation is done just once.
- **Consistency can be ensured before starting search**
  - Algorithm MAC (Maintaining Arc Consistency)
    - AC is checked before search and after each assignment
- It is possible to use stronger consistency techniques (e.g. use them once before starting search).

Variable ordering in labelling influences significantly efficiency of constraint solvers (e.g. in a tree-structured CSP).

Which variable ordering should be chosen in general?

FAIL FIRST principle

„select the variable whose instantiation will lead to a failure“

it is better to tackle failures earlier, they can be become even harder

- **prefer the variables with smaller domain** (dynamic order)
  - a smaller number of choices ~ lower probability of success
  - the dynamic order is appropriate only when new information appears during solving (e.g., in look ahead algorithms)

„solve the hard cases first, they may become even harder later“

- **prefer the most constrained variables**
  - it is more complicated to label such variables (it is possible to assume complexity of satisfaction of the constraints)
  - this heuristic is used when there is an equal size of the domains
- **prefer the variables with more constraints to past variables**
  - a static heuristic that is useful for look-back techniques

Order of values in labelling influences significantly efficiency (if we choose the right value each time, no backtrack is necessary).

What value ordering for the variable should be chosen in general?

SUCCEED FIRST principle

„prefer the values belonging to the solution“

- if no value is part of the solution then we have to check all values
- if there is a value from the solution then it is better to find it soon

Note: SUCCEED FIRST does not go against FAIL FIRST !

- **prefer values with more supports**
  - this information can be found in AC-4
- **prefer a value leading to less domain reduction**
  - this information can be computed using singleton consistency
- **prefer a value simplifying the problem**
  - solve approximation of the problem (e.g. a tree)

Generic heuristics are usually too complex for computation.

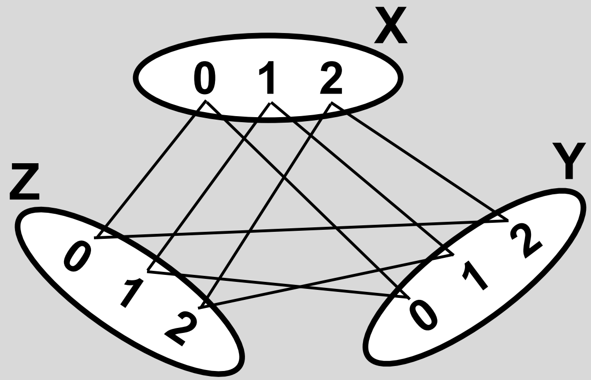
It is better to use problem-driven heuristics that propose the value!

So far we assumed search by labelling, i.e. assignment of values to variables.

- assign a value, propagate and backtrack in case of failure (try other value)
  - this is called **enumeration**
- propagation is used only after instantiating a variable

Example:

- X,Y,Z in 0,…,N-1 (N is constant)
- X=Y, X=Z, Z=(Y+1) mod N
  - problem is AC, but has no solution
  - enumeration will try all the values
  - for n=10^7 runtime 45 s. (at 1.7 GHz P4)



Can we use faster labelling?

Enumeration resolves disjunctions in the form $\mathbf{X=0} \vee \mathbf{X=1} \ldots \mathbf{X=N-1}$

- if there is no correct value, the algorithm tries all the values

We can use propagation when we find some value to be wrong!

- that value is deleted from the domain which starts propagation that filters out other values
- we solve disjunctions in the form $\mathbf{X=H} \vee \mathbf{X \neq H}$
- this is called **step labelling** (usually a default strategy)
- the previous example solved in 22 s. by trying and refuting value 0 for X

  Why so long?

  - In each AC cycle we remove just one value.

Another typical branching is **bisection/domain splitting**

- we solve disjunctions in the form $\mathbf{X \leq H} \vee \mathbf{X > H}$, where H is a value in the middle of the domain

Search and heuristics

When solving real-life problems we frequently have some experience with "manual" solving of the problem.

Heuristics – a guide where to go

- they recommend a value for assignment (value ordering)
- frequently lead to a solution

But what to do when the heuristic is wrong?

- DFS takes care about the end of branches (leafs of tree)
- it repairs latest failures of the heuristic rather than earlier failures
- so it assumes that heuristic was right at the beginning of search

Observation1:

The number of wrong heuristic decisions is **low**.

Observation2:

Heuristics are usually **less reliable at the beginning** of search than at its end (more information and fewer choices are available there).

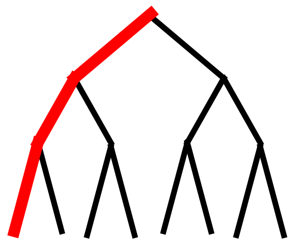
How to make search more efficient?

– Backtracking is “blind” with respect to heuristics.

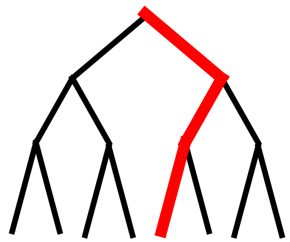
Discrepancy = violation of heuristic (different value is used)

Core principles of discrepancy search:

– we change the order of branches based on discrepancies
– explore first the branches with **less discrepancies**

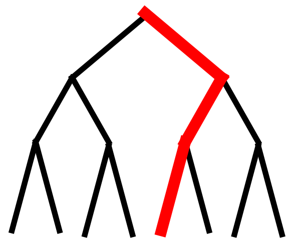


is before

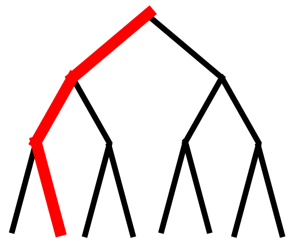


heuristic says „go left“

– explore first the branches with **earlier discrepancies**



is before



heuristic says „go left“

Limited Discrepancy Search

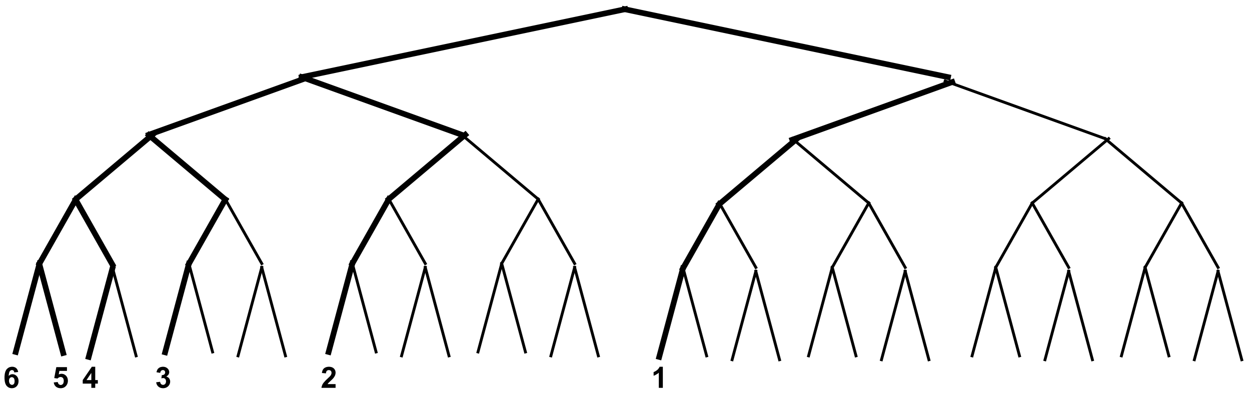
Limited number of discrepancies (cutoff)

- branches with less discrepancies are explored first

After failure **increase the number of allowed discrepancies** by one (restart).

- first, follow the heuristic
- then explore paths with at most one discrepancy

Example: LDS(1), heuristic suggests going to left



A note for **non-binary domains**:

- non-heuristic values are assumed as **one discrepancy** (here)
- each other non-heuristic value means **increase of the number of discrepancies** (e.g. third value = two discrepancies)

```
procedure LDS-PROBE(Unlabelled,Labelled,Constraints,D)
    if Unlabelled = {} then return Labelled
    select X in Unlabelled
    Values_X ← D_X - {values inconsistent with Labelled using Constraints}
    if Values_X = {} then return fail
    else select HV in Values_X using heuristic
        if D>0 then
          for each value V from Values_X-{HV} do
            R ← LDS-PROBE(Unlabelled-{X}, Labelled ∪ {X/V}, Constraints, D-1)
            if R≠ fail then return R
          end for
        end if
        return LDS-PROBE(Unlabelled-{X}, Labelled∪{X/HV}, Constraints, D)
    end if
end LDS-PROBE
procedure LDS(Variables,Constraints)
    for D=0 to |Variables| do   % D determines the allowed number of discrepancies
        R ← LDS-PROBE(Variables,{},Constraints,D)
        if R≠ fail then return R
    end for
    return fail
end LDS
```

So far we looked for any solution satisfying the constraints.

Frequently, we need to find an optimal solution, where solution quality is defined by some objective function.

Definition:

- **Constraint Satisfaction Optimisation Problem** (CSOP) consists of a CSP P and an objective function *f* mapping solutions of P to real numbers.
- A **solution to a CSOP** is a solution to P minimizing / maximizing the value of *f*.
- When solving CSOPs we need methods that can provide more than one solution.

The method **branch-and-bound** is a frequently used optimisation technique based on pruning branches where there is no optimal solution.

It uses a **heuristic function** h that estimates the value of objective function f.

- admissible heuristic for minimization satisfies $h(x) \leq f(x)$
  [for maximization $f(x) \leq h(x)$]
- heuristic closer to f is better

We stop exploring the search branch when:

- there is **no solution** in the sub-tree
- there is **no optimal solution** in the sub-tree
  - Bound $\leq h(x)$, where Bound is the maximal value of f for an acceptable solution

How to obtain the Bound?

- for example the value of the solution found so far

Objective function is encoded in a constraint

we "optimize" the value v, where v = f(x)

- the first solution is found using no bound on v
- the next solutions must be better than the last solution found (v < Bound)
- repeat until no feasible solution is found

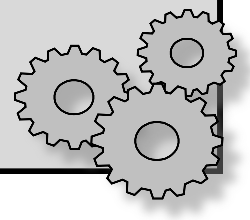
Algorithm Branch & Bound

```
procedure BB-Min(Variables, V, Constraints)
    Bound ← sup
    NewSolution ← fail
    repeat
        Solution ← NewSolution
        NewSolution ← Solve(Variables,Constraints ∪ {V<Bound})
        Bound ← value of V in NewSolution (if any)
    until NewSolution = fail
    return Solution
end BB-Min
```

Branch and bound: notes

- Heuristic h is hidden in the **propagation of constraint** v = f(x).
- Efficiency of search depends on:
  - **good heuristic** (good propagation through the objective constraint)
  - **good solution found early**
    using an initial bound may help
- We can find the optimal solution fast
  - but the **proof of optimality takes time** (explore the rest of search tree)
- Frequently, we do not need optimal solution, good solution is enough
  - BB can stop after finding a good enough solution
- BB can be speeded up by using both upper and lower bounds

```
repeat
    TempBound ← (UBound+LBound) / 2
    NewSolution ← Solve(Variables,Constraints ∪ {V≤TempBound})
    if NewSolution=fail then
        LBound ← TempBound+1
    else
        UBound ← TempBound
until LBound = UBound
```



How Is CSP used in practice?

Exploiting the principles of constraint satisfaction, but **programming them ad-hoc** for a given problem.

- flexibility (complete customisation to a given problem)
- speed (for a given problem)
- expensive in terms of initial development and maintenance

Exploiting an **existing constraint solver**.

- usually integrated to a host language as a library
- contains core constraint satisfaction algorithms
- the user can focus on problem modelling
- It is hard to modify low-level implementation (domains,…)
- sometimes possible to implement own constraints
- frequently possible to implement own search strategies

A typical structure of constraint models:

```
declare_variables( Variables),
post_constraints( Variables),
labeling( Variables ).
```

Definition of variables and their domains

Definition of constraints

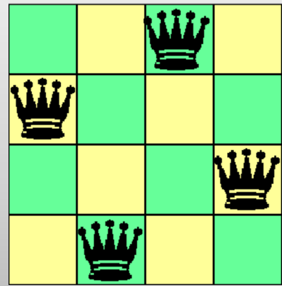
Declarative model

Control part
- **exploration of space of assignments**
- **assigning values to variables**
- **looking for one, all, or optimal solution**

N-queens

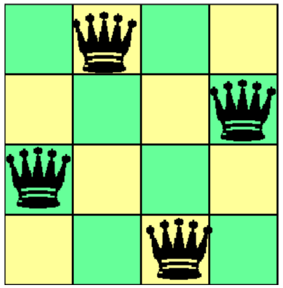
Propose a constraint model for solving the **N-queens problem** (place N queens to a chessboard of size NxN such that there is no conflict).

```
Variables: X1,…,Xn, Y1,…,Yn
Domain: 1,…,N
Constraints:
   all_different({X1,…,Xn}),
   all_different({Y1,…,Yn}),
   ∀i<j: |Xi - Xj| \= |Yi - Yj|
```



Solutions (for 4 queens) in the form (X_i,Y_i)

```
[(1,2),(2,4),(3,1),(4,3)]
[(1,3),(2,1),(3,4),(4,2)]
[(1,2),(2,4),(4,3),(3,1)]
[(1,3),(2,1),(4,2),(3,4)]
[(1,2),(3,1),(2,4),(4,3)]
[(1,3),(3,4),(2,1),(4,2)]
[(1,2),(3,1),(4,3),(2,4)]
[(1,3),(3,4),(4,2),(2,1)]
…
```



Where is the problem?

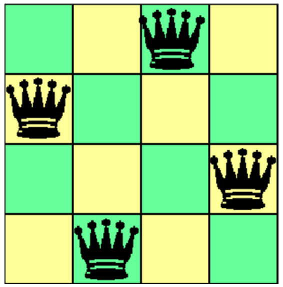
- Different assignments describe the same solution!
- There are only two different solutions (very „similar“ solutions).
- The search space is non-necessarily large.

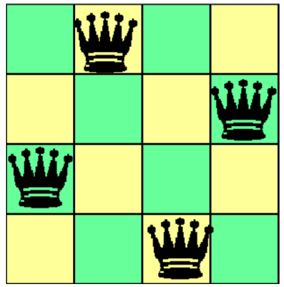
Pre-assign queens to columns, use only variables for rows

Variables: $X_1, \ldots, X_n$
Domain: $1, \ldots, N$
Constraints:
  `all_different`$(\{X_1, \ldots, X_n\})$,
  $\forall i<j$: $|X_i - X_j|$ `\=` $j-i$

Solutions (for 4 queens) in the form X_i values

`[2,4,1,3]`
`[3,1,4,2]`



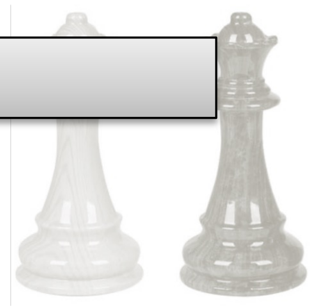


Model properties:

- fewer variables (= smaller state space)
- fewer constraints (= faster propagation)

Remove symmetrical solutions:

X_1 `=< ceiling(N/2)`

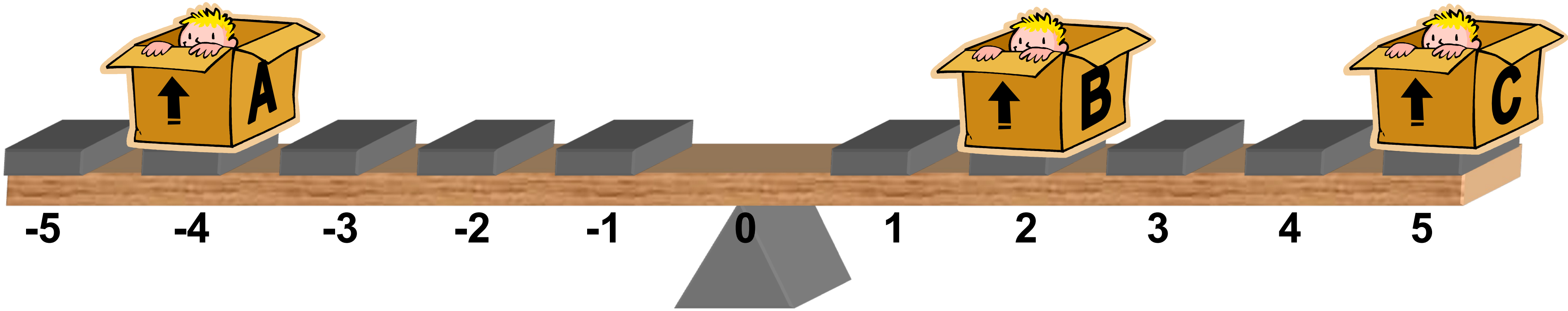


a so-called **symmetry breaking constraint**

Seesaw problem

The problem:

Adam (36 kg), Boris (32 kg) and Cecil (16 kg)
want to sit on a seesaw with the length 10 foots
such that the minimal distances between them are more than 2 foots and the seesaw is balanced.



A constraint model:

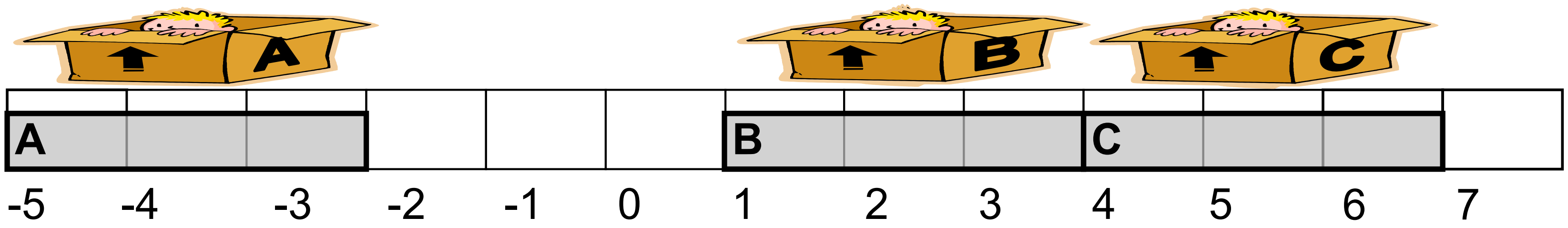
`A,B,C in -5..5`	position
`36*A+32*B+16*C = 0`	equilibrium state
`|A-B|>2, |A-C|>2, |B-C|>2`	minimal distances

Seesaw problem: a different perspective

```
A,B,C in -5..5,
A =< 0,
36*A+32*B+16*C = 0,
abs(A-B)>2,
abs(A-C)>2,
abs(B-C)>2
```

```
A in -4..0
B in -1..5
C in -5..5
```

- A set of similar constraints typically indicates a structured sub-problem that can be represented using a **global constraint.**



- We can use a global constraint describing **allocation of activities to exclusive resource**.

```
A,B,C in -5..5,
A =< 0,
36*A+32*B+16*C = 0,
cumulative([task(A,3,_,1,1),task(B,3,_,1,2),
            task(C,3,_,1,3)],[limit(1)]),
```

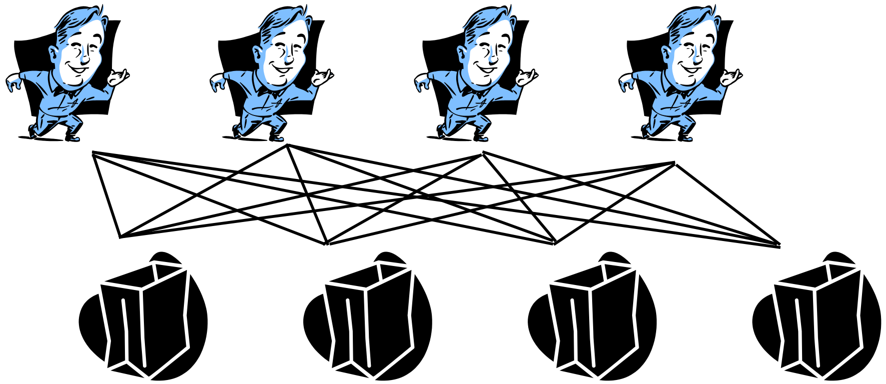
task(start,duration,end,capacity,id)

```
A in -4..0
B in -1..5
C in (-5..-3) \/ (-1..5)
```

Assignment problem

The problem:

There are 4 workers and 4 products and a table describing the efficiency of producing the product by a given worker. The task is assign workers to products (one to one) in such a way that the total efficiency is at least 19.



	P1	P2	P3	P4
W1	7	1	3	4
W2	8	2	5	1
W3	4	3	7	2
W4	3	1	6	3

A constraint model:

`W1,W2,W3,W4 in 1..4`	a product per worker
`all_different([W1,W2,W3,W4])`	different products
$T_{1,W1}+T_{2,W2}+T_{3,W3}+T_{4,W4} \geq 19$	total efficiency

Assignment problem - a dual model

Why do we **assign products to workers?**

Cannot we do it in an opposite way, that is, to **assign a worker to a product?**

Of course, we can **swap the role of values and variables!**

- This new model is called a **dual model.**

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

assignment_dual(Products):-
    Products = [P1,P2,P3,P4],

    domain(Products,1,4),
    all_different(Products),
    element(P1,[7,8,4,3],EP1),
    element(P2,[1,2,3,1],EP2),
    element(P3,[3,5,7,6],EP3),
    element(P4,[4,1,2,3],EP4),
    EP1+EP2+EP3+EP4 #>= 19,

    labeling([ff],Products).
```

element(X,List,Y) ⇔ $List_X = Y$

```
P1 in 1..2
P2 in 1..4
P3 in 2..4
P4 in 1..4
```

	Number of choice points
Primal model	**15**
Dual model	**11**

Which model is better?

- In this particular case, the dual model propagates earlier (thus it is assumed to be better).

We can combine both primal and dual model in a single model to get better domain pruning.

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

assignment_combined(Workers):-
    Workers= [W1,W2,W3,W4],
    domain(Workers,1,4),
    all_different(Workers),
    element(W1,[7,1,3,4],EW1),
    element(W2,[8,2,5,1],EW2),
    element(W3,[4,3,7,2],EW3),
    element(W4,[3,1,6,3],EW4),
    EW1+EW2+EW3+EW4 #>= 19,

    Products = [P1,P2,P3,P4],
    domain(Products,1,4),
    all_different(Products),
    element(P1,[7,8,4,3],EP1),
    element(P2,[1,2,3,1],EP2),
    element(P3,[3,5,7,6],EP3),
    element(P4,[4,1,2,3],EP4),
    EP1+EP2+EP3+EP4 #>= 19,

    assignment(Workers,Products),

    labeling([ff],Workers).
```

a primal model

```
W1 in (1..2)\/{4}
W2 in 1..4
W3 in 2..4
W4 in 2..4
```

a dual model (redundant)

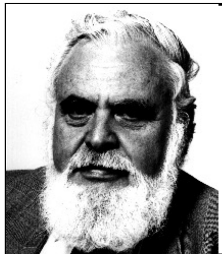
```
P1 in 1..2
P2 in 1..4
P3 in 2..4
P4 in 1..4
```

a channelling constraint

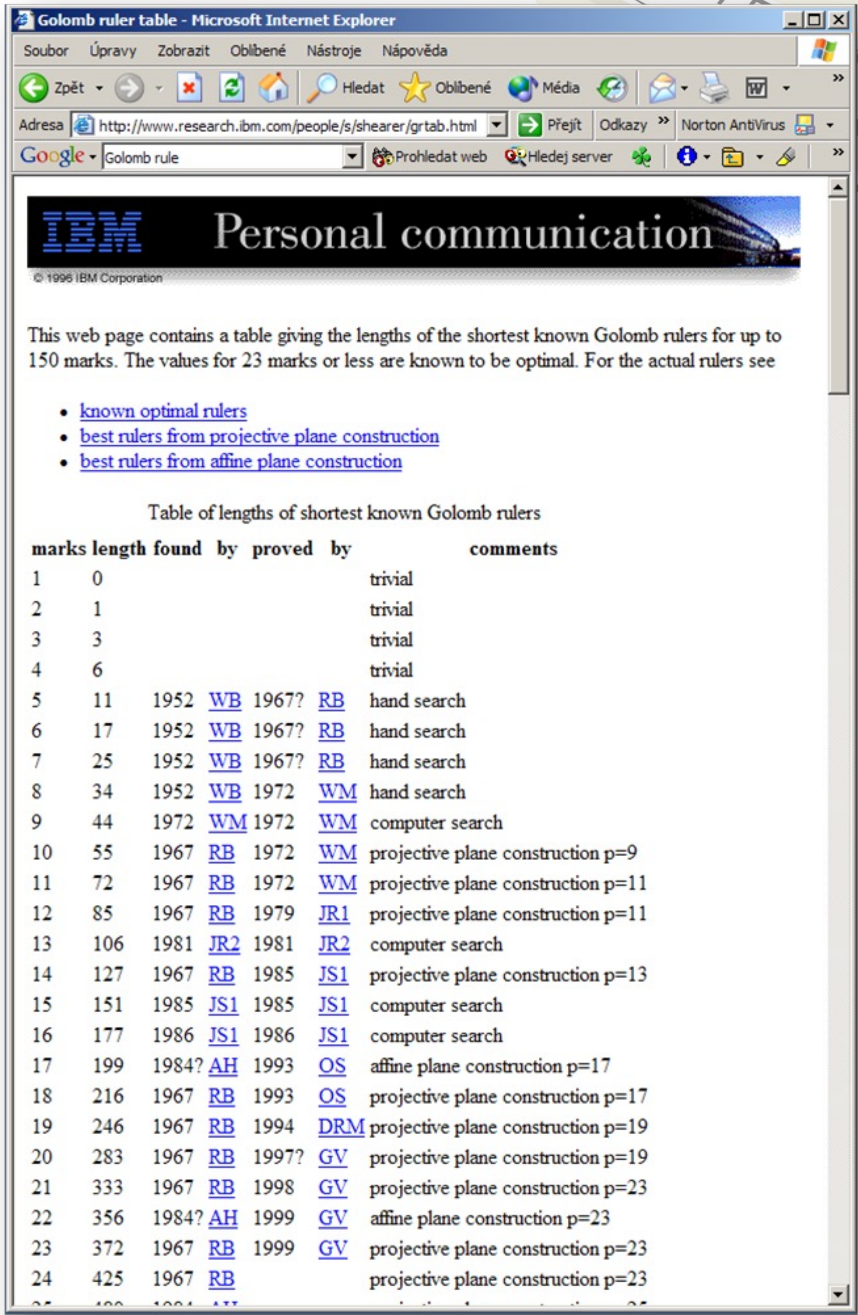
labelling one model is enough

- A **ruler with M marks** such that **distances** between any two marks are **different**.
- The **shortest ruler** is the optimal ruler.

- **Hard** for M≥16, no exact algorithm for M ≥ 24!
- Applied in **radioastronomy**.



Solomon W. Golomb
Professor
University of Southern California
http://csi.usc.edu/faculty/golomb.html

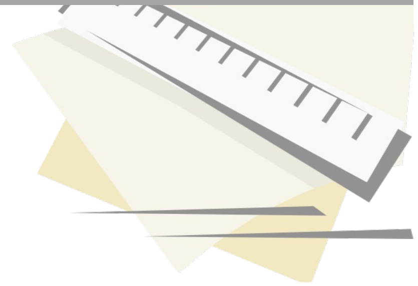


This web page contains a table giving the lengths of the shortest known Golomb rulers for up to 150 marks. The values for 23 marks or less are known to be optimal. For the actual rulers see

- known optimal rulers
- best rulers from projective plane construction
- best rulers from affine plane construction

Table of lengths of shortest known Golomb rulers

marks	length	found	by	proved	by	comments
1	0					trivial
2	1					trivial
3	3					trivial
4	6					trivial
5	11	1952	WB	1967?	RB	hand search
6	17	1952	WB	1967?	RB	hand search
7	25	1952	WB	1967?	RB	hand search
8	34	1952	WB	1972	WM	hand search
9	44	1972	WM	1972	WM	computer search
10	55	1967	RB	1972	WM	projective plane construction p=9
11	72	1967	RB	1972	WM	projective plane construction p=11
12	85	1967	RB	1979	JR1	projective plane construction p=11
13	106	1981	JR2	1981	JR2	computer search
14	127	1967	RB	1985	JS1	projective plane construction p=13
15	151	1985	JS1	1985	JS1	computer search
16	177	1986	JS1	1986	JS1	computer search
17	199	1984?	AH	1993	OS	affine plane construction p=17
18	216	1967	RB	1993	OS	projective plane construction p=17
19	246	1967	RB	1994	DRM	projective plane construction p=19
20	283	1967	RB	1997?	GV	projective plane construction p=19
21	333	1967	RB	1998	GV	projective plane construction p=23
22	356	1984?	AH	1999	GV	affine plane construction p=23
23	372	1967	RB	1999	GV	projective plane construction p=23
24	425	1967	RB			projective plane construction p=23



A base model:

Variables X_1, …, X_M with the domain 0..M*M

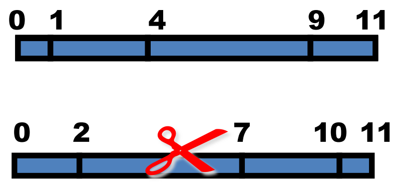
$X_1 = 0$ *ruler start*

$X_1 < X_2 < \ldots < X_M$ *no permutations of variables*

$\forall i<j\ D_{i,j} = X_j - X_i$ *difference variables*

all_different($\{D_{1,2}, D_{1,3}, \ldots D_{1,M}, D_{2,3}, \ldots D_{M,M-1}\}$)

Model extensions:



$D_{1,2} < D_{M-1,M}$ *symmetry breaking*

better bounds (**implied constraints**) for $D_{i,j}$

$$D_{i,j} = D_{i,i+1} + D_{i+1,i+2} + \ldots + D_{j-1,j}$$

so **$D_{i,j} \geq \Sigma_{j-i} = (j-i)*(j-i+1)/2$** *lower bound*

$$X_M = X_M - X_1 = D_{1,M} = D_{1,2} + D_{2,3} + \ldots D_{i-1,i} + D_{i,j} + D_{j,j+1} + \ldots + D_{M-1,M}$$

$$D_{i,j} = X_M - (D_{1,2} + \ldots D_{i-1,i} + D_{j,j+1} + \ldots + D_{M-1,M})$$

so **$D_{i,j} \leq X_M - (M-1-j+i)*(M-j+i)/2$** *upper bound*

Golomb ruler - some results

- **What is the effect of different constraint models?**

size	base model	base model + symmetry	base model + symmetry + implied constraints
7	220	80	30
8	1 462	611	190
9	13 690	5 438	1 001
10	120 363	49 971	7 011
11	2 480 216	985 237	170 495

time in milliseconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM

- **What is the effect of different search strategies?**

size	fail first			leftmost first		
	enum	*step*	*bisect*	*enum*	*step*	*bisect*
7	40	60	40	30	30	30
8	390	370	350	220	190	200
9	2 664	2 384	2 113	1 182	1 001	921
10	20 870	17 545	14 982	8 782	7 011	6 430
11	1 004 515	906 323	779 851	209 251	170 495	159 559

time in milliseconds on Mobile Pentium 4-M 1.70 GHz, 768 MB RAM

Constraint satisfaction is a technology for **declarative solving combinatorial (optimization) problems**.

Constraint modeling

- describing problems as constraint satisfaction problems (variables, domains, constraints)

Constraint satisfaction

- local search techniques
- combination of depth-first search with inference (constraint propagation/consistency techniques)
- ad-hoc algorithms encoded in global constraints

It is easy to model problems in terms of a CSP

… but it is complicated to design solvable models.

Charles University, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz