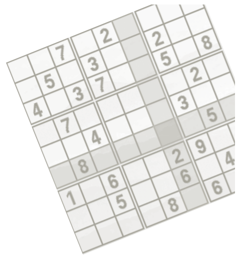


Constraint (Logic) Programming

Roman Barták

Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic

bartak@ktiml.mff.cuni.cz

Combinatorial puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

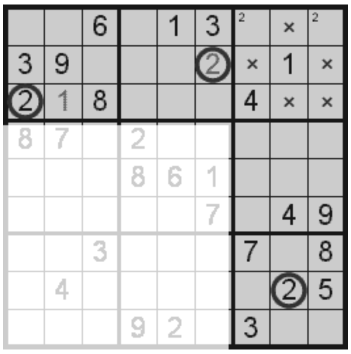
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Solving Sudoku

×	×	6		1	3			
3	9	×					1	
2	1	8				4		

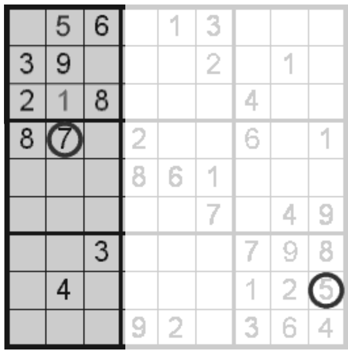
Use information that each digit appears exactly once in each row, column and sub-grid.

Solving Sudoku

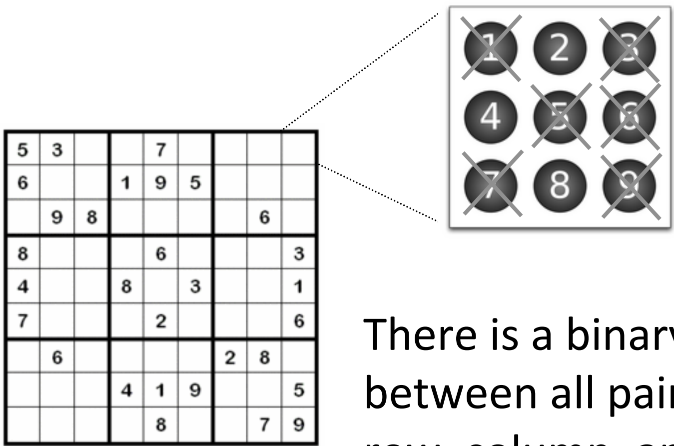


- If neither rows and columns provide enough information, we can note allowed digits in each cell.

- The position of a digit can be inferred from positions of other digits and restrictions of Sudoku that each digit appears one in a column (row, sub-grid)



Sudoku in general



We can see every cell as a **variable** with possible values from **domain** {1,…,9}.

There is a binary inequality **constraint** between all pairs of variables in every row, column, and sub-grid.

Such formulation of the problem is called a **constraint satisfaction problem.**

- **Constraint satisfaction in general**
  - search techniques (backtracking)
  - consistency techniques (arc consistency)
  - global constraints (all-different)
  - combing search and consistency
    - value and variable ordering
    - branching schemes
  - optimization problems
- **Constraints in Logic Programming**
  - from unification to constraints
  - constraints in SICStus Prolog
  - modeling examples

Constraint satisfaction problem consists of:

- a finite set of **variables**
  - describe some features of the world state that we are looking for, for example positions of queens at a chessboard
- **domains** – finite sets of values for each variable
  - describe "options" that are available, for example the rows for queens
  - sometimes, there is a single common "superdomain" and domains for particular variables are defined via unary constraints
- a finite set of **constraints**
  - a constraint is a *relation* over a subset of variables
for example rowA ≠ rowB
  - a constraint can be defined *in extension* (a set of tuples satisfying the constraint) or using a *formula* (see above)

A feasible solution of a constraint satisfaction problem is a complete consistent assignment of values to variables.

- **complete** = each variable has assigned a value
- **consistent** = all constraints are satisfied

Sometimes we may look for all the feasible solutions or for the number of feasible solutions.

An optimal solution of a constraint satisfaction problem is a feasible solution that minimizes/maximizes a value of some objective function.

- **objective function** = a function mapping feasible solutions to real numbers

- **Problem Modelling**
  How to describe a problem as a constraint satisfaction problem?

- **Solving Techniques**
  How to find values for the variables satisfying all the constraints?

Solving CSPs

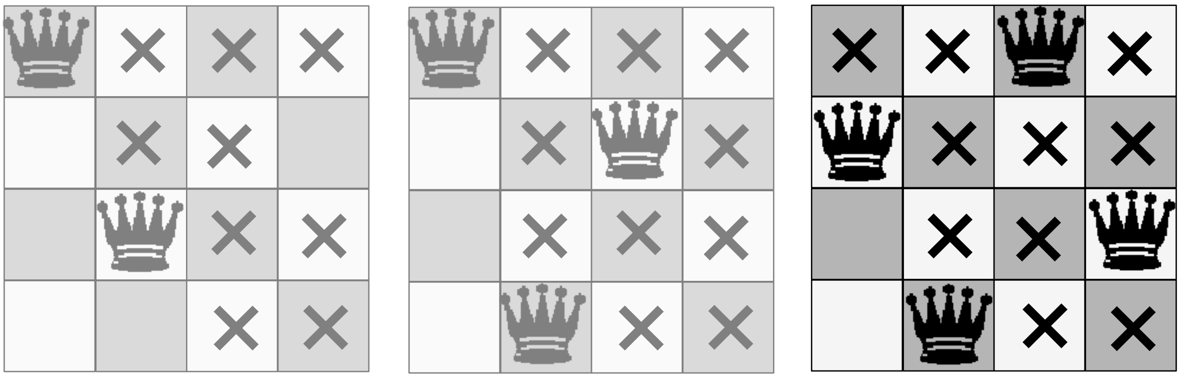
N-queens: allocate N queens to a chess board of size N×N in a such way that no two queens attack each other

the modelling decision: each queen is located in its own column

variables: N variables r(i) with the domain {1,…,N}

constraints: no two queens attack each other

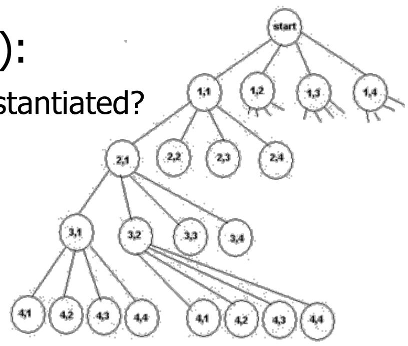
$$\forall i \neq j \quad r(i) \neq r(j) \wedge |i-j| \neq |r(i)-r(j)|$$



Backtracking

- Probably the most widely used systematic search algorithm that **verifies the constraints as soon as possible**.
  - upon failure (any constraint is violated) the algorithm goes back to the last instantiated variable and tries a different value for it
  - depth-first search
- The core principle of applying backtracking to solve a CSP:
  1. assign values to variables one by one
  2. after each assignment verify satisfaction of constraints with known values of all constrained variables

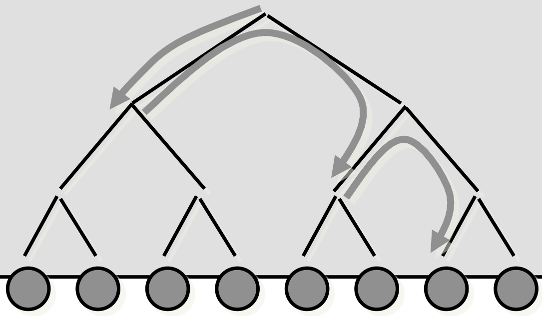
  Open questions (to be answered later):
  - What is the order of variables being instantiated?
  - What is the order of values tried?



- **Backtracking explores partial consistent assignments until it finds a complete (consistent) assignment.**

Chronological Backtracking (a recursive version)

```
procedure BT(X:variables, V:assignment, C:constraints)
    if X={} then return V
    x ← select a not-yet assigned variable from X
    for each value h from the domain of x do
       if constraints C are consistent with V ∪ {x/h} then
           R ← BT(X – {x}, V ∪ {x/h}, C)
           if R ≠ fail then return R
    end for
    return fail
end BT

Call as BT(X, {}, C)
```



Note:

If it is possible to perform the test stage for a partially generated solution candidate then BT is always better than GT, as BT does not explore all complete solution candidates.

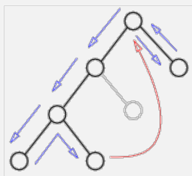
Weaknesses of Backtracking

Look Back

- **thrashing**
  - throws away the reason of the conflict

  Example: A,B,C,D,E:: 1..10, A>E
  - BT tries all the assignments for B,C,D before finding that A≠1

  Solution: **backjumping** (jump to the source of the failure)



- **redundant work**
  - unnecessary constraint checks are repeated

  Example: A,B,C,D,E:: 1..10, B+8<D, C=5*E
  - when labelling C,E the values 1,..,9 are repeatedly checked for D

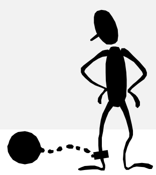
  Solution: **backmarking**, **backchecking** (remember (no-)good assignments)

Look Ahead

- **late detection of the conflict**
  - constraint violation is discovered only when the values are known

  Example: A,B,C,D,E::1..10, A=3*E
  - the fact that A>2 is discovered when labelling E

  Solution: **forward checking** (forward check of constraints)

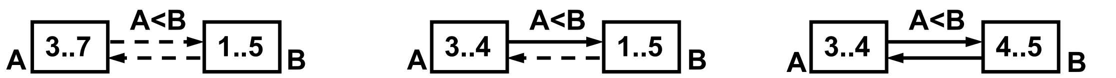


Constraint consistency

Example:

A in [3,..,7], B in [1,..,5], A<B

Constraint can be used to **prune the domains** actively using a dedicated filtering algorithm!



Not arc-consistent | **(A,B) is consistent** | **(A,B) and (B,A) are consistent**

Some definitions:

The arc (Vi,Vj) is **arc consistent** iff for each value x from the domain Di there exists a value y in the domain Dj such that the assignment Vi =x a Vj = y satisfies all the binary constraints on Vi, Vj.

CSP is **arc consistent** iff every arc (Vi,Vj) is arc consistent (in both directions).

Algorithm for arc revisions

How to make (V_i,V_j) arc consistent?

- Delete all the values *x* from the domain D_i that are inconsistent with all the values in D_j (there is no value *y* in D_j such that the valuation V_i = x, V_j = y satisfies all the binary constrains on V_i a V_j).

Algorithm of arc revision

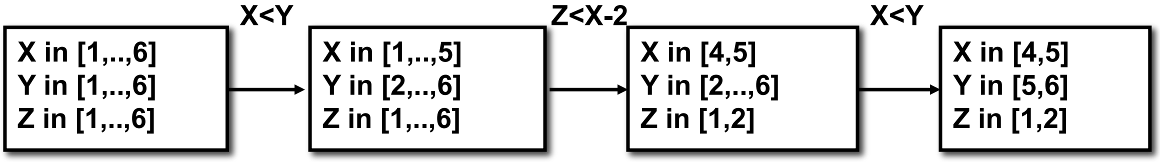
procedure REVISE((i,j))
    DELETED ← false
    for each X in D_i **do**
        if there is no such Y in D_j such that (X,Y) is consistent, i.e., (X,Y) satisfies all the constraints on V_i, V_j **then**
            delete X from D_i
            DELETED ← true
        end if
    end for
    return DELETED
end REVISE

The procedure also reports the deletion of some value.

Constraint propagation

How to establish arc consistency among the constraints?

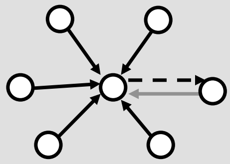
Example: X in [1,..,6], Y in [1,..,6], Z in [1,..,6], X<Y, Z<X-2



Make all the constraints consistent until any domain is changed (AC-1)

Why we should revise the constraint X<Y if domain of Z is changed?

```
procedure AC-3(G)
    Q ← {(i,j) | (i,j)∈arcs(G), i≠j}          % queue of arcs for revision
    while Q non empty do
        select and delete (k,m) from Q
        if REVISE((k,m)) then
            Q ← Q ∪ {(i,k) | (i,k)∈arcs(G), i≠k, i≠m}
        end if
    end while
end AC-3
```



Non-binary constraints

So far we assumed mainly **binary constraints**.

We can use binary constraints, because **every CSP can be converted to a binary CSP**!

Is this really done in practice?

- in many applications, non-binary constraints are naturally used, for example, a+b+c ≤ 5
- for such constraints we can do some local inference / propagation

  for example, if we know that a,b ≥ 2, we can deduce that c ≤ 1
- within a single constraint, we can restrict the domains of variables to the values satisfying the constraint

  ➯ **generalized arc consistency**

Generalized arc consistency

- **The value** x of variable V is **generalized arc consistent** with respect to constraint P if and only if there exist values for the other variables in P such that together with x they satisfy the constraint P

  Example: A+B≤C, A in {1,2,3}, B in {1,2,3}, C in {1,2,3}
  Value 1 for C is not GAC (it has no support), 2 and 3 are GAC.

- **The variable** V is **generalized arc consistent** with respect to constraint P, if and only if all values from the current domain of V are GAC with respect to P.

  Example: A+B≤C, A in {1,2,3}, B in {1,2,3}, C in {2,3}
  C is GAC, A and B are not GAC

- **The constraint** C is **generalized arc consistent**, if and only if all variables in C are GAC.

  Example: for A in {1,2}, B in {1,2}, C in {2,3} A+B≤C is GAC

- **The constraint satisfaction problem** P is **generalized arc consistent**, if and only if all the constraints in P are GAC.

How to make a CSP GAC?

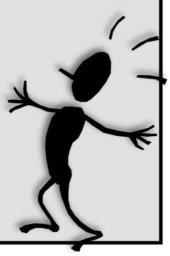
We will modify AC-3 for non-binary constraints.

- We can see a constraint as a set of propagation methods – each method makes one variable GAC:

  A + B = C: A + B → C, C – A → B, C – B → A

- By executing all the methods we make the constraint GAC.
- We repeat revisions until any domain changes.

```
procedure GAC-3(G)
    Q ← {Xs →Y | Xs →Y is a method for some constraint in G}
    while Q non empty do
        select and delete (As→B) from Q
        if REVISE(As→B) then
            if D_B=∅ then stop with fail
            Q ← Q ∪ {Xs →Y | Xs →Y is a method s.t. B∈Xs}
        end if
    end while
end GAC-3
```



- Can we achieve GAC **faster than a general GAC algorithm**?
  - for example revision of A < B can be done much faster via bounds consistency.
- Can we write a **filtering algorithm for a constraint** whose **arity varies**?
  - for example all_different constraint
- We can exploit **semantics of the constraint** for efficient filtering algorithms that can work with any number of variables.

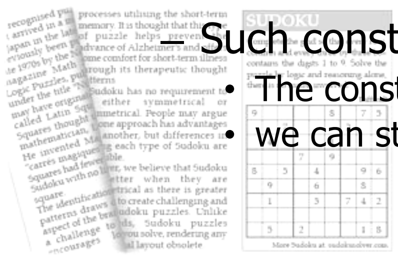
☞ **global constraints** ☜

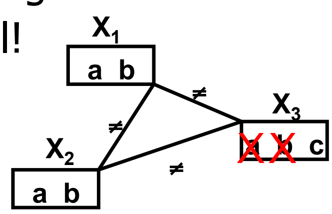
- **Logic-based puzzle,** whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

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

How to model such a problem?

- variables **describe the cells**
- **inequality constraint** connect each pair of variables in each row, column, and sub-grid
- Such constraints do not propagate well!
  - The constraint network is AC, but
  - we can still remove some values.





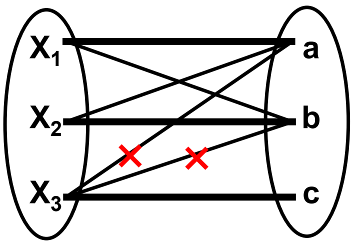
Régin (AAAI 1994)

all-different

This constraint models a complete set of binary inequalities.

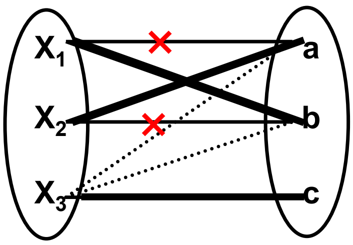
`all_different`$(\{X_1,\dots, X_k\}) = \{(d_1,\dots, d_k) \mid \forall i\ \ d_i \in D_i\ \ \&\ \forall i \neq j\ \ d_i \neq d_j\}$

Domain filtering is based on **matching in bipartite graphs**
(nodes = variables+values, edges = description of domains)



Initialization:

1) find a maximum matching
2) remove all edges that do not belong to any maximum matching



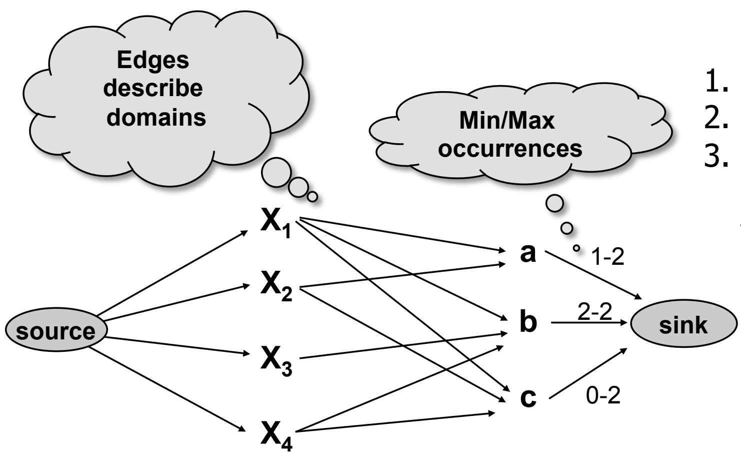
Incremental propagation ($X_1 \neq a$):

1) remove "deleted" edges
2) find a new maximum matching
3) remove all edges that do not belong to any maximum matching

Régin (AAAI 1996)

global cardinality

- A generalization of all-different
  - the number of occurrences of a value in a set of variables is restricted by minimal and maximal numbers of occurrences
- Efficient filtering is based on **network flows.**



1. make a value graph
2. add sink and source
3. set upper and lower bounds and edge capacities (0-1 and value occurrences)

A maximal flow corresponds to a feasible assignment of variables!
We will find edges with zero flow in each maximal flow and the we will remove the corresponding edges.

How to solve constraint satisfaction problems?

- So far we have two methods:
  - **search**
    - complete (finds a solution or proves its non-existence)
    - too slow (exponential)
      - explores "visibly" wrong valuations
  - **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!

Maintaining consistency during search

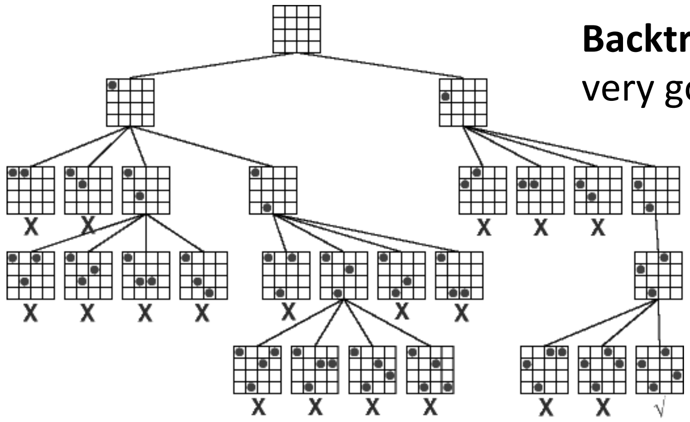
A core constraint satisfaction method:

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

Look-ahead technique (MAC – Maintaining Arc Consistency)

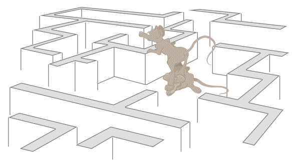
```
procedure labeling(V, D, C)
        if all variables from V are instantiated then return V
        select not-yet instantiated variable x from V
        for each value v from Dx do
                (TestOK, D') ← consistent(V, D, C∪{x=v})
                if TestOK=true then R ← labeling(V, D', C)
                        if R ≠ fail then return R
        end for
        return fail
end labeling
```

Is a CSP solved by enumeration?



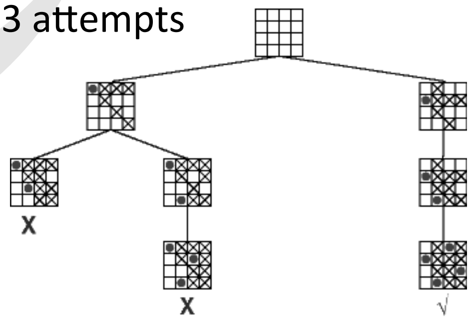
Backtracking (enumeration) is not very good

- 19 attempts



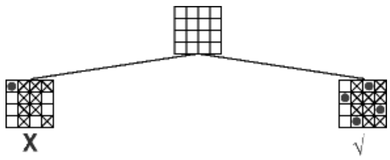
Forward checking is better

3 attempts



And the winner is **Look Ahead**

2 attempts



Variable ordering

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

Value ordering

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

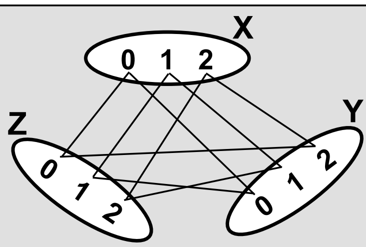
Generic heuristics are usually too complex for computation.

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

Enumeration

- 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?

Other branching strategies

- **Enumeration** resolves disjunctions in the form **X=0** ∨ **X=1 … X=N-1**
  - if there is no correct value, the algorithm tries all the values
- **We can use propagation when we find some value is wrong!**
  - that value is deleted from the domain which starts propagation that filters out other values
  - we solve disjunctions in the form **X=H** ∨ **X≠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 **X≤H** ∨ **X>H**, where H is a value in the middle of the domain

Constrained optimization

- So far we looked for any solution satisfying the constraints.
- Frequently, we need to find an optimal solution, where solution quality is defined by an 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.

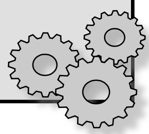
Objective function is encoded in a constraint

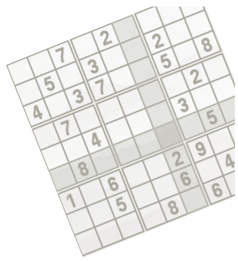
we „optimize" a value v, where v = f(x)

- 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
```





Constraints in Prolog

Practical Exercises

Prolog „program" consists of **rules** and **facts**.

Each **rule** has the structure **Head:-Body.**

- **Head** is a (compound) term
- **Body** is a query (a conjunction of terms)
  - typically Body contains all variables from Head
- rule semantics: **if Body is true then Head can be deduced**

Fact can be seen as a rule with an empty (true) body.

Query is a conjunction of terms: Q = **Q1,Q2,…,Qn.**

- **Find a rule** whose head matches goal Q1.
  - If there are more rules then introduce a choice point and use the first rule.
  - If no rule exists then backtrack to the last choice point and use an alternative rule there.
- **Use the rule body** to substitute Q1.
  - For facts (Body=true), the goal Q1 disappears.
- **Repeat until empty query** is obtained.

Prolog = Unification + Backtracking

- **Unification** (matching)
  - to select an appropriate rule
  - to compose an answer substitution
  - How?
    - make the terms syntactically identical by applying a substitution
- **Backtracking** (depth-first search)
  - to explore alternatives
  - How?
    - resolve the first goal (from left) in a query
    - apply the first applicable rule (from top)

Unification?

Recall:

```
?-3=1+2.
no
?-X=1+2
X=1+2;
no
?-3=X+1
no
```

What is the problem?

Term has no meaning (even if it consists of numbers), it is just a syntactic structure!

We would like to have:

```
?-X=1+2.
X=3

?-3=X+1.
X=2

?-3=X+Y,Y=2.
X=1

?-3=X+Y,Y>=2,X>=1.
X=1
Y=2
```

Constraint Logic Programming

- For each variable we define its **domain.**
  - we will be using discrete finite domains only
  - such domains can be mapped to integers
- We define **constraints/relations** between the variables.

```
?-domain([X,Y],0,100),3#=X+Y,Y#>=2,X#>=1.
```

- Recall a **constraint satisfaction problem.**
- We want the system to find the values for the variables in such a way that all the constraints are satisfied.

```
X=1, Y=2
```

How does it work?

How is constraint satisfaction realized?

– For each variable the system keeps its actual domain.
– When a constraint is added, the inconsistent values are removed from the domain.

Example:

	X	**Y**
	`inf..sup`	`inf..sup`
`domain([X,Y],0,100)`	`0..100`	`0..100`
`3#=X+Y`	`0..3`	`0..3`
`Y#>=2`	`0..1`	`2..3`
`X#>=1`	`1`	`2`

SEND+MORE=MONEY

Assign different digits to letters such that SEND+MORE=MONEY holds and S≠0 and M≠0.

Idea:

generate assignments with different digits and check the constraint

6.8 s

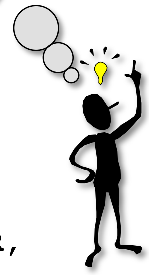
```
solve_naive([S,E,N,D,M,O,R,Y]):-
   Digits1_9 = [1,2,3,4,5,6,7,8,9],
   Digits0_9 = [0|Digits1_9],
   member(S, Digits1_9),
   member(E, Digits0_9), E\=S,
   member(N, Digits0_9), N\=S, N\=E,
   member(D, Digits0_9), D\=S, D\=E, D\=N,
   member(M, Digits1_9), M\=S, M\=E, M\=N, M\=D,
   member(O, Digits0_9), O\=S, O\=E, O\=N, O\=D, O\=M,
   member(R, Digits0_9), R\=S, R\=E, R\=N, R\=D, R\=M, R\=O,
   member(Y, Digits0_9), Y\=S, Y\=E, Y\=N, Y\=D, Y\=M, Y\=O, Y\=R,
              1000*S + 100*E + 10*N + D +
              1000*M + 100*O + 10*R + E =:=
   10000*M + 1000*O + 100*N + 10*E + Y.
```

equality of arithmetic expressions

SEND+MORE=MONEY (better)

```
solve_better([S,E,N,D,M,O,R,Y]):-
   Digits1_9 = [1,2,3,4,5,6,7,8,9],
   Digits0_9 = [0|Digits1_9],
   % D+E = 10*P1+Y
   member(D, Digits0_9),
   member(E, Digits0_9), E\=D,
   Y is (D+E) mod 10, Y\=D, Y\=E,
   P1 is (D+E) // 10, % carry bit

   % N+R+P1 = 10*P2+E
   member(N, Digits0_9), N\=D, N\=E, N\=Y,
   R is (10+E-N-P1) mod 10, R\=D, R\=E, R\=Y, R\=N,
   P2 is (N+R+P1) // 10,

   % E+O+P2 = 10*P3+N
   O is (10+N-E-P2) mod 10, O\=D, O\=E, O\=Y, O\=N, O\=R,
   P3 is (E+O+P2) // 10,

   % S+M+P3 = 10*M+O
   member(M, Digits1_9), M\=D, M\=E, M\=Y, M\=N, M\=R, M\=O,
   S is 9*M+O-P3,
   S>0,S<10, S\=D, S\=E, S\=Y, S\=N, S\=R, S\=O, S\=M.
```

Some letters can be computed from other letters and invalidity of the constraint can be checked before all letters are know

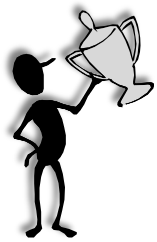


0 s

SEND+MORE=MONEY (CLP)

Domain filtering can take care about computing values for letters that depend on other letters.

0 s



```
:-use_module(library(clpfd)).
solve(Sol):-
  Sol=[S,E,N,D,M,O,R,Y],
  domain([E,N,D,O,R,Y],0,9),
  domain([S,M],1,9),
            1000*S + 100*E + 10*N + D +
            1000*M + 100*O + 10*R + E #=
  10000*M + 1000*O + 100*N + 10*E + Y,
  all_different([S,E,N,D,M,O,R,Y]),
  labeling([],Sol).
```

assign values (from domains) to variables – depth first search

Note: It is also possible to use a model with carry bits.

- A typical structure of CLP programs:

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

solve(Sol):-
   declare_variables( Variables),
   post_constraints( Variables),
   labeling( Variables ).
```

Definition of CLP operators, constraints and solvers

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**

- **Domain** in SICStus Prolog is a set of integers
  - other values must be mapped to integers
  - integers are naturally ordered
- frequently, domain is an interval
  - `domain(ListOfVariables,MinVal,MaxVal)`
  - defines variables with the initial domain {MinVal,…,MaxVal}
- For each variable we can define a separate domain (it is possible to use union, intersection, or complement)
  - `X in MinVal..MaxVal`
  - `X in (1..3) \/ (5..8) \/ {10}`

Internal representation of domains

- Each domain is represented as a list of disjoint intervals
  - $[[Min_1|Max_1],[Min_2|Max_2],\ldots,[Min_n|Max_n]]$
  - $Min_i \leq Max_i < Min_{i+1} - 1$
- Domain definition is like a unary constraint
  - if there are more domain definitions for a single variable then their intersection is used (like the conjunction of unary constraints)

```
?-domain([X],1,20), X in 15..30.
X in 15..20
```

Arithmetic constraints

- Classical arithmetic constraints with operations +,-, *, /, abs, min, max,... all operations are built-in
- It is possible to use comparison to define a constraint #=, #<, #>, #=<, #>=, #\=

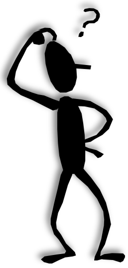
```
?-A+B #=< C-2.
```

- What if we define a constraint before defining the domains?
  - For such variables, the system assumes initially the infinite domain inf..sup

Arithmetic (reified) constraints can be connected using logical operations:

- `#\ :Q` negation
- `:P #/\ :Q` conjunction
- `:P #\ :Q` exclusive disjunction („exactly one“)
- `:P #\/ :Q` disjunction
- `:P #=> :Q` implication
- `:Q #<= :P` implication
- `:P #<=> :Q` equivalence

```
?- X#<5 #\/ X#>7.
X in inf..sup
```



Let us start with a simple example

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

a(X):- X#<5.
a(X):- X#>7.
```

```
?- a(X).
 {SYNTAX ERROR: in
 line 3 (within 3-4)}
 ** operator expected
 after expression **
no
```

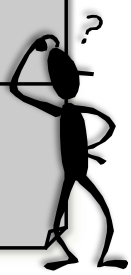
What is the problem?

The constraint model is disjunctive, i.e., we need to backtrack to get the model where X>7!

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

a(X):- X#<5 #\/ X#>7.
```

```
?- a(X).

X in inf..sup ? ;

no
```

```
?- a(X), X#>5.

X in 8..sup ? ;

no
```



The propagator waits until all but one component of the disjunction are proved to fail and then it propagates through the remaining component.

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

a(X):- X in (inf..4) \/ (8..sup).
```

```
?- a(X).

X in (inf..4)\/(8..sup) ? ;

no
```

Constructive disjunction

How does it work in general?

$a_1(X) \vee a_2(X) \vee \ldots a_n(X)$

– **propagate** each constraint $a_i(X)$ **separately**

– **union** all the restricted **domains** for X

This could be an expensive process!

Actually, it is close to **singleton consistency**:

- X in 1..5 ⇒ X=1 ∨ X=2 ∨ X=3 ∨ X=4 ∨ X=5

We can still write special propagators for particular disjunctive constraints!

- Constraints alone frequently do not set the values to variables. We need instantiate the variables via search.
- `indomain(X)`
  - assign a value to variable X (values are tried in the increasing order upon backtracking)
- `labeling(Options,Vars)`
  - instantiate variables in the list Vars
  - algorithm MAC – maintaining arc consistency during backtracking

labeling(:Options, +Variables)

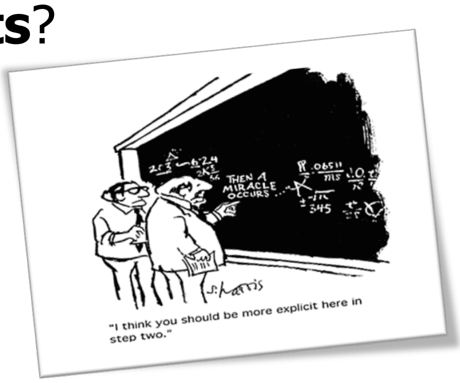
- variable ordering
  - `leftmost` (default), `min`, `max`, `ff`, `ffc`
  - `variable(Sel)`, where `Sel` is a name of own procedure for variable selection - `Sel(Vars,Selected,Rest)`
- value ordering
  - `step` (default), `enum`, `bisect`
  - `up` (default), `down`
  - `value(Enum)`, where `Enum` is a name of own procedure for value selection - `Enum(X,Rest,BB0,BB)`
- rest
  - `all`, `minimize(X)`, `maximize(X)`
  - `discrepancy(D)`

- Find all solutions to the equality $A + B = 10$ for $A, B \in \{1, 2, . . . , 10\}$

```
:- use_module(library(clpfd)).
arithmetics(A,B) :-
  domain([A,B], 1, 10),
  A + B #= 10,
  labeling([],[A,B]).
```

Problem modelling

- Which **decision variables** are needed?
  - variables denoting the problem solution
  - they also define the search space
- Which **values** can be assigned to variables?
  - the definition of domains influences the constraints used
- How to formalise **constraints**?
  - available constraints
  - auxiliary variables may be necessary



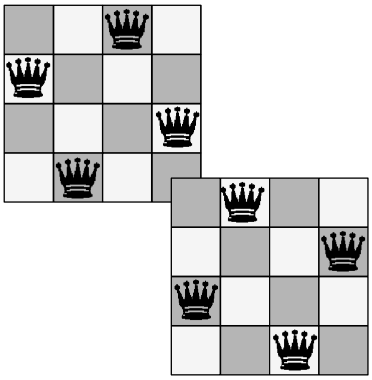
4-queens

- Propose a constraint model for solving the 4-queens problem (place four queens to a chessboard of size 4x4 such that there is no conflict).

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

queens([(X1,Y1),(X2,Y2),(X3,Y3),(X4,Y4)]):-
   Rows = [X1,X2,X3,X4], Columns = [Y1,Y2,Y3,Y4],
   domain(Rows,1,4),
   domain(Columns,1,4),
   all_different(Rows), all_different(Columns),
   abs(X1-X2) #\= abs(Y1-Y2),
   abs(X1-X3) #\= abs(Y1-Y3), abs(X1-X4) #\= abs(Y1-Y4),
   abs(X2-X3) #\= abs(Y2-Y3), abs(X2-X4) #\= abs(Y2-Y4),
   abs(X3-X4) #\= abs(Y3-Y4),
   append(Rows,Columns, Variables),
   labeling([], Variables).
```

4-queens: analysis

```
?- queens(L).
L = [(1,2),(2,4),(3,1),(4,3)] ? ;
L = [(1,3),(2,1),(3,4),(4,2)] ? ;
L = [(1,2),(2,4),(4,3),(3,1)] ? ;
L = [(1,3),(2,1),(4,2),(3,4)] ? ;
L = [(1,2),(3,1),(2,4),(4,3)] ? ;
L = [(1,3),(3,4),(2,1),(4,2)] ? ;
L = [(1,2),(3,1),(4,3),(2,4)] ? ;
L = [(1,3),(3,4),(4,2),(2,1)] ? ;
…
```



Where is the problem?

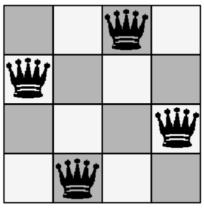
- Different assignments describe the same solution!
- There are only two different solutions (very „similar" solutions).
- The search space is non-necessarily large.

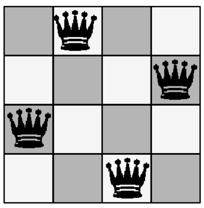
Solution

- pre-assign queens to rows (or to columns)

4-queens: a better model

```
:-use_module(library(clpfd)).
queens4(Queens):-
   Queens = [X1,X2,X3,X4],
   domain(Queens,1,4),
   all_different(Queens),
   abs(X1-X2) #\= 1, abs(X1-X3) #\= 2, abs(X1-X4) #\= 3,
   abs(X2-X3) #\= 1, abs(X2-X4) #\= 2,
   abs(X3-X4) #\= 1,
   labeling([], Queens).

?- queens4(Q).
Q = [2,4,1,3] ? ;
Q = [3,1,4,2] ? ;
no
```





Model properties:

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

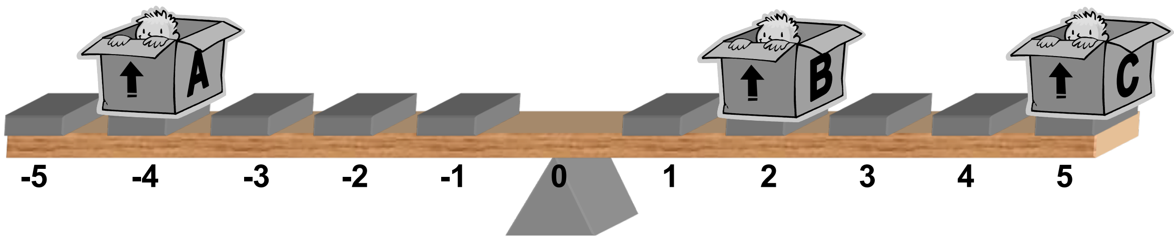
Homework:

- Write a constraint model for arbitrary number of queens (given as input)
- think about further improvements

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 CSP 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 - implementation

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

seesaw(Sol):-
    Sol = [A,B,C],

    domain(Sol,-5,5),
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

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

```
?- seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;
X = [4,-5,1] ? ;
X = [4,-4,-1] ? ;
X = [4,-2,-5] ? ;

no
```

Symmetry breaking

– important to reduce search space

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

seesaw(Sol):-
    Sol = [A,B,C],

    domain(Sol,-5,5),
    A #=< 0,
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

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

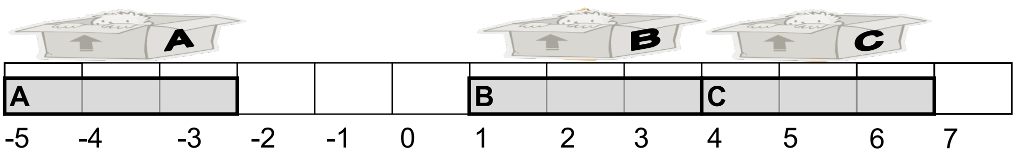
```
?- seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;

no
```

Seesaw problem - a different perspective

```
domain([A,B,C],-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**.

```
domain([A,B,C],-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)]),
```

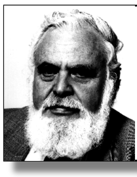
task(start,duration,end,capacity,id)

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

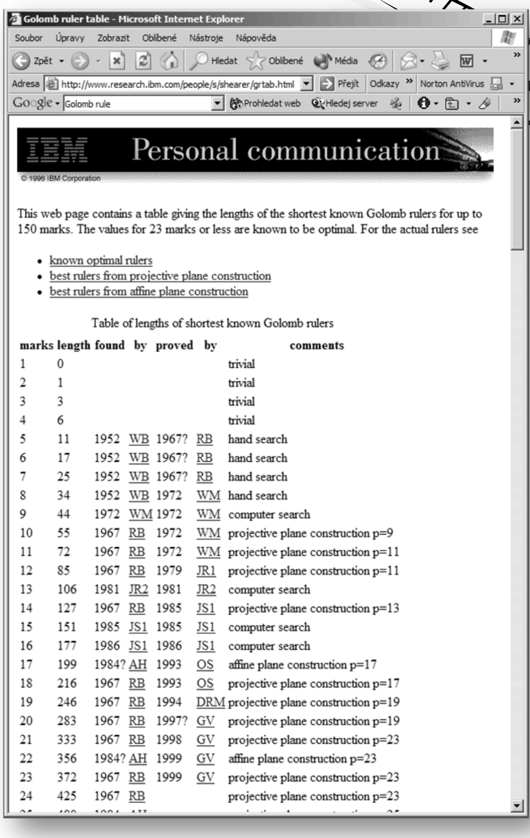
Golomb ruler

- 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



Golomb ruler table - Microsoft Internet Explorer

http://www.research.ibm.com/people/s/shearer/grtab.html

IBM Personal communication

© 1996 IBM Corporation

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

Golomb ruler – a model

A base model:

Variables $X_1, \ldots, X_M$ with the domain 0..M*M

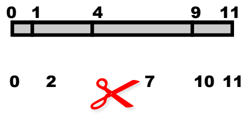
$X_1 = 0$ *ruler start*

$X_1 < X_2 < \ldots < X_M$ *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

Sky Observatory

- Assume a sky observatory with four **telescopes**:
  - Newton, Kepler, Dobson, Monar
- Each day, each telescope is used by one of the following **observers**:
  - scientists (3), students (2), visitors (1), nobody (0)
- Each day, we know the expected **weather**
  - ideal (0), worse (1), no-observations (2)
- and **phases of the moon**
  - 0 (new moon), ..., 4 (full moon), 5, 6.
- The **problem input** is defined by two lists (of equal length) of weather and moon conditions:
  - [1,1,0,0,1,2,1,0],
  - [1,1,2,2,3,3,4,4]

Sky Observatory - restrictions

- Newton and Kepler cannot be used together.
- Newton cannot be used by visitors.
- Scientists are never using Monar.
- Dobson cannot be used around full moon (3-5).
- Scientists (students) use at most one telescope each day.
- Students must use at least one telescope during the whole scheduling period.
- When the weather is ideal either students or scientists must use some telescope.

Sky Observatory - objectives

- Using each telescope costs some money (**expenses**), and visitors pay some money (**income**) for using the telescope according to the following table:

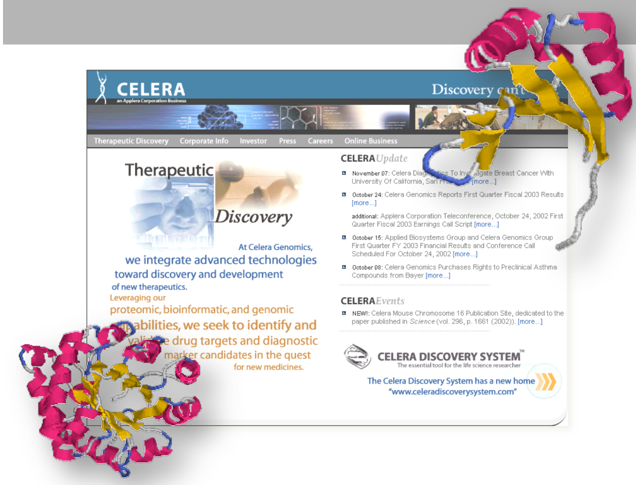
	Monar	Dobson	Kepler	Newton
expenses	10	50	60	70
income	20	60	100	100

- In case of bad weather or bad moon conditions (3-5) there is 50% **discount** for visitors when using Monar or Dobson.
- There is some **initial budget** given and the final **balance cannot be negative.**
- **Maximize scientific output** of observations (scientists are preferred over students that are preferred over the visitors).

Sky Observatory - constraint model

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

observatory(Moon,Weather,Budget, Schedule):-
        days(Moon,Weather,Budget, Schedule),
        append(Schedule, Vars),
        count(2,Vars,#>,0),
               % students must use at least one telescope during the whole scheduling period
        sum(Vars,#=,Obj),
               % scientists are preferred over students that are preferred over the visitors
        labeling([max,maximize(Obj)],Vars).

days([],[],Budget, []):- Budget #>= 0.
days([M|Moon],[W|Weather],Budget, [S|Schedule]):-
        S = [Newton, Kepler, Dobson, Monar],
        (W = 2 -> domain(S,0,0) ; domain(S,0,3)), % bad weather- > non observations
        Newton#=0 #\/ Kepler#=0,        % Newton and Kepler cannot be used together
        Newton #\= 1,                   % Newton cannot be used by visitors
        Monar #\= 3,                    % scientists are never using Monar
        ((3=<M, M=<5) -> Dobson#=0 ; true), % Dobson cannot be used around full moon (3-5)
        global_cardinality(S,[0-Nobody,1-Visitors,2-Students,3-Scientists]),
        Scientists #=<1, Students #=< 1,
               % scientists (students) use at most one telescope each day
        (W=0 -> Scientists+Students #>0 ; true),
          % when the weather is ideal either students or scientists must use some telescope
        table([[Monar,ME,MI]], [[0,0,0],[1,10,20],[2,10,0],[3,10,0]]),
        table([[Dobson,DE,DI]], [[0,0,0],[1,50,60],[2,50,0],[3,50,0]]),
        table([[Kepler,KE,KI]], [[0,0,0],[1,60,100],[2,60,0],[3,60,0]]),
        table([[Newton,NE,NI]], [[0,0,0],[1,70,100],[2,70,0],[3,70,0]]),
        (((3=<M, M=<5); W=1) ->
          % bad weather or bad moon conditions -> 50% discount for Monar or Dobson
             NextBudget #= Budget-ME-DE-KE-NE+MI/2+DI/2+KI+NI
        ;    NextBudget #= Budget-ME-DE-KE-NE+MI+DI+KI+NI),
        !, days(Moon,Weather,NextBudget, Schedule).
```

Some Real Applications



Bioinformatics

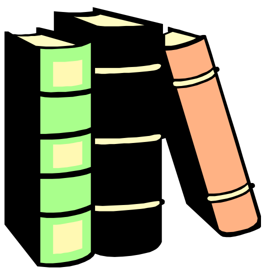
- DNA sequencing (Celera Genomics)
- deciding the 3D structure of proteins from the sequence of amino acids

Planning and Scheduling

- automated planning of spacecraft activities (Deep Space 1)
- manufacturing scheduling

Resources

- **Books**
  - P. Van Hentenryck: **Constraint Satisfaction in Logic Programming**, MIT Press, 1989
  - E. Tsang: **Foundations of Constraint Satisfaction**, Academic Press, 1993
  - K. Marriott, P.J. Stuckey: **Programming with Constraints: An Introduction**, MIT Press, 1998
  - R. Dechter: **Constraint Processing**, Morgan Kaufmann, 2003
  - **Handbook of Constraint Programming**, Elsevier, 2006
- **Journals**
  - **Constraints**, An International Journal. Springer Verlag
  - **Constraint Programming Letters**, free electronic journal
- **On-line resources**
  - **Course Web** (transparencies) http://ktiml.mff.cuni.cz/~bartak/podminky/
  - **On-line Guide to Constraint Programming** (tutorial) http://ktiml.mff.cuni.cz/~bartak/constraints/
  - **Constraints Archive** (archive and links) http://4c.ucc.ie/web/archive/index.jsp
  - **Constraint Programming online** (community web) http://www.cp-online.org/



Roman Barták
Charles University in Prague, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz