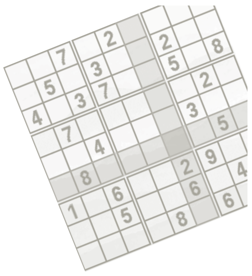
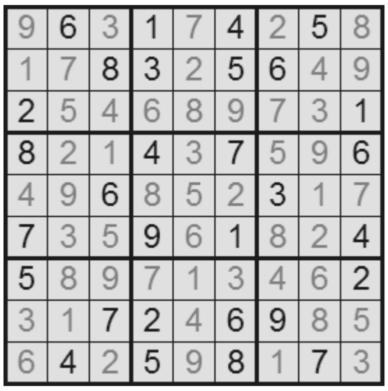


Constraint (Logic) Programming

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

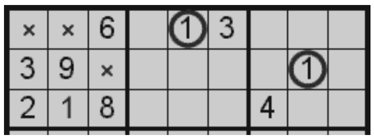
Sudoku

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

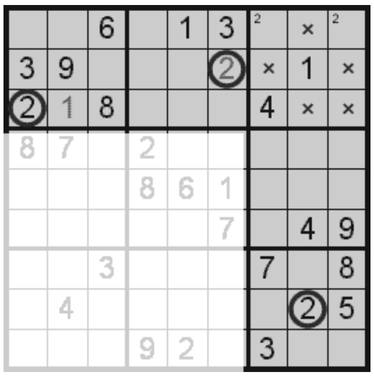
Solving Sudoku



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

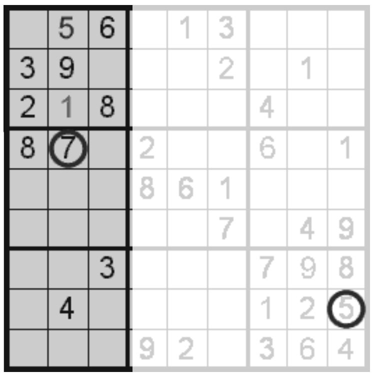
Use information that each digit appears exactly once
in each row, column and sub-grid.

Solving Sudoku

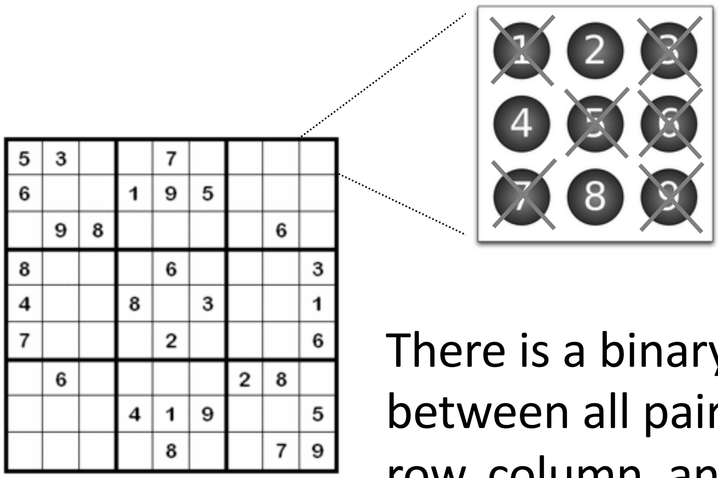


If neither rows and columns provide enough information, we can note allowed digits in each cell.

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.**

Tutorial outline

Constraint satisfaction in general

- search techniques (backtracking)
- consistency techniques (arc consistency)
- global constraints (all-different)
- combining search and consistency
  - value and variable ordering
  - branching schemes
- optimization problems

Constraints in Logic Programming

- from unification to constraints
- constraints in Picat
- modeling examples

Constraint Satisfaction Problem

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 solution to a CSP

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 integers

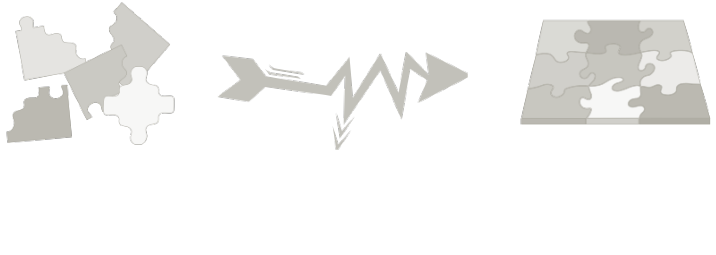
The Core Topics

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

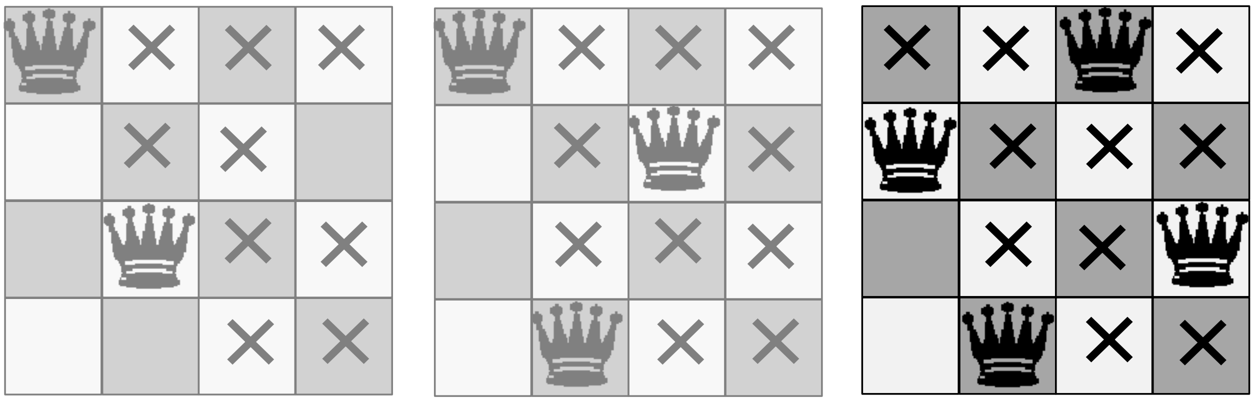
N-queens: allocate N queens to a chess board of size N×N in a such way that no two queens attack each other

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

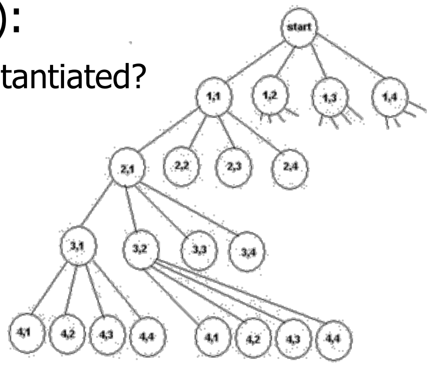
The core principle of applying backtracking to solve a CSP:

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):

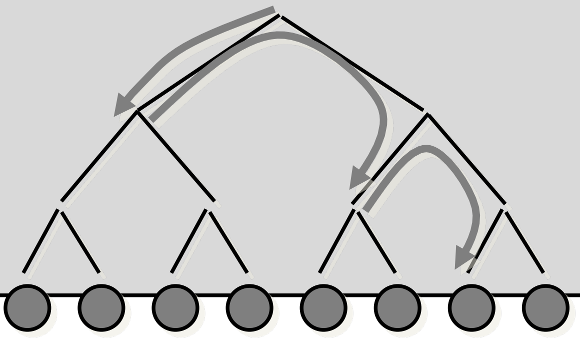
- What is the order of variables being instantiated?
- What is the order of values tried?

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

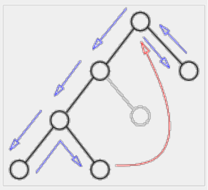
thrashing

– 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

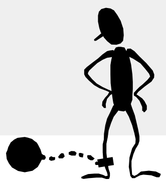
late detection of the conflict

– 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)

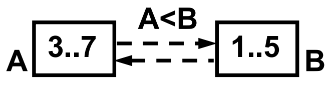


Constraint consistency

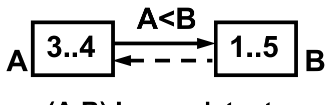
Example:

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

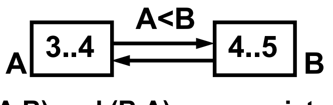
Constraint can be used to **prune the domains** actively using a dedicated filtering algorithm!



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

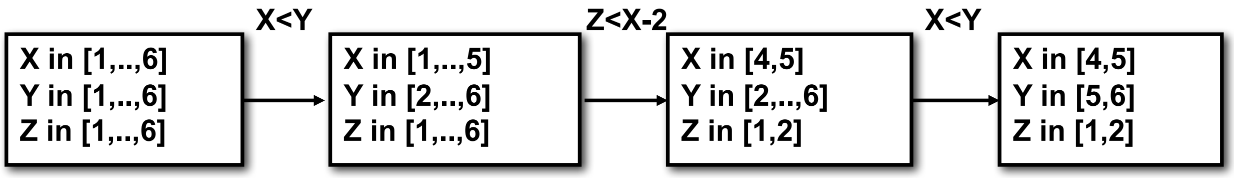
```
procedure REVISE((i,j))
    DELETED ← false
    for each X in $D_i$ do
        if there is no such Y in $D_j$ such that (X,Y) is consistent, i.e.,
                (X,Y) satisfies all the constraints on $V_i$, $V_j$ then
            delete X from $D_i$
            DELETED ← true
        end if
  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

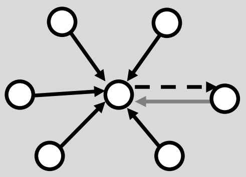
Example: X in [1,..,6], Y in [1,..,6], Z in [1,..,6], X<Y, Z<X-2



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, $a+b+c \leq 5$

– for such constraints we can do some local inference / propagation

for example, if we know that $a,b \geq 2$, we can deduce that $c \leq 1$

– 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.

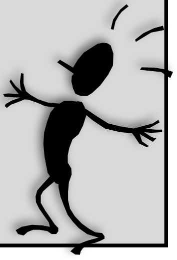
How to make a CSP GAC?

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:

  $A + B = C$: $A + B \rightarrow C$, $C - A \rightarrow B$, $C - B \rightarrow A$
- 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
```



Global constraints

Can we achieve GAC **faster than a general GAC algorithm**?

- for example revision of A < B can be done much faster via bounds consistency.

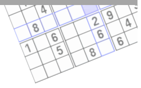
Can we write a **filtering algorithm for a constraint** whose **arity varies**?

- for example all_different constraint

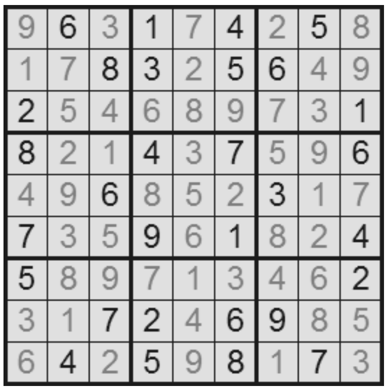
We can exploit **semantics of the constraint** for efficient filtering algorithms that can work with any number of variables.

☞ **global constraints** ☜

Recall Sudoku



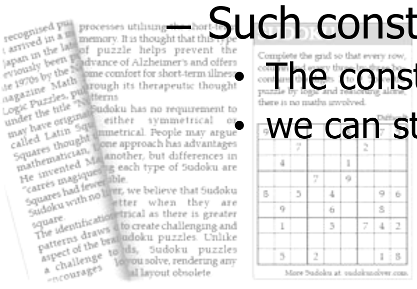
Logic-based puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

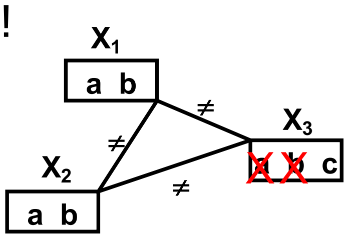


9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

How to model such a problem?

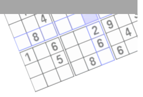
- variables **describe the cells**
- **inequality constraint** connect each pair of variables in each row, column, and sub-grid
- Such constraints do not propagate well!
  - The constraint network is AC, but
  - we can still remove some values.





Régin (AAAI 1994)

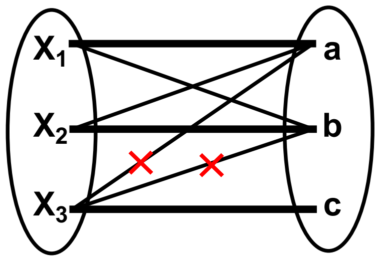
all-different



This constraint models a complete set of binary inequalities.

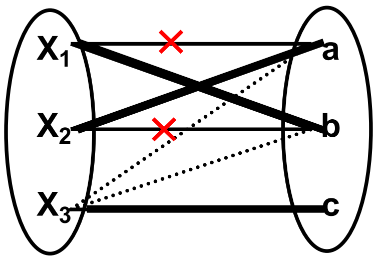
`all_different`($\{X_1,\dots, X_k\}$) = $\{(d_1,\dots, d_k) \mid \forall i\ \ d_i \in D_i\ \ \&\ \forall i \neq j\ \ d_i \neq d_j\}$

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

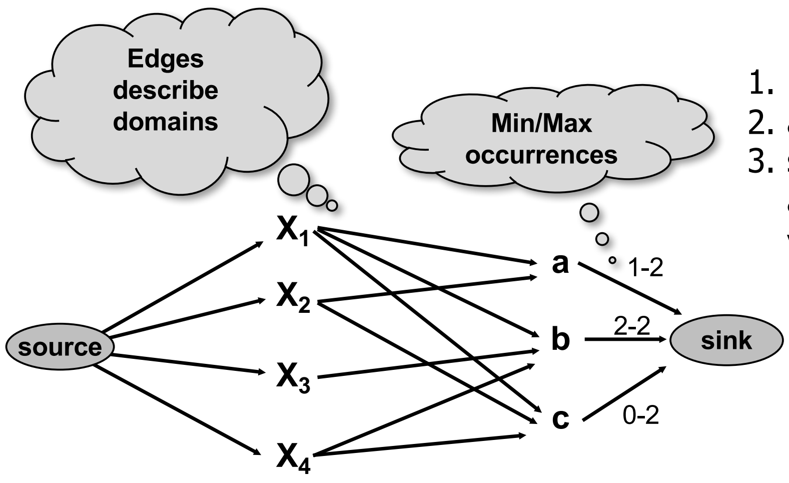
Régin (AAAI 1996)

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

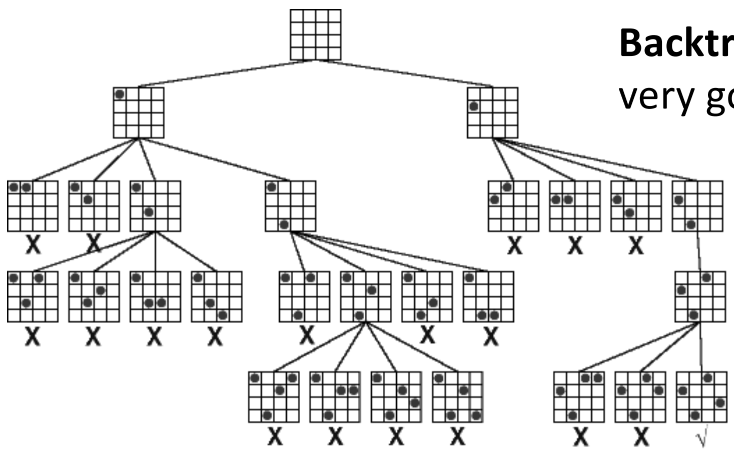
A core constraint satisfaction method:

- **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)

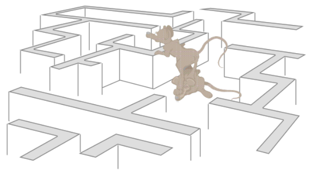
```
procedure labeling(V, D, C)
    if all variables from V are instantiated then return V
    select not-yet instantiated variable x from V
    for each value v from $D_x$ do
        (TestOK, D’) ← consistent(V, D, C∪{x=v})
        if TestOK=true then R ← labeling(V, D’, C)
            if R ≠ fail then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

Is a CSP solved by enumeration?



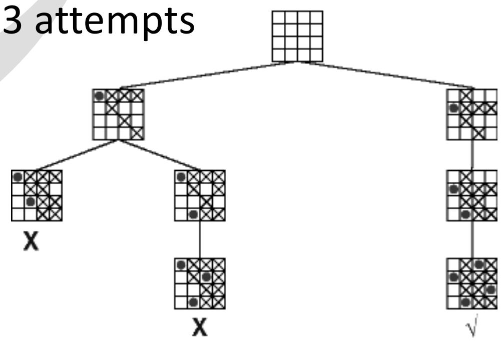
Backtracking (enumeration) is not very good

- 19 attempts



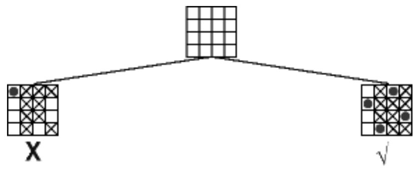
Forward checking is better

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!

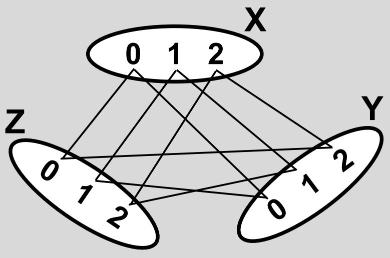
Enumeration

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.

Branch and bound for constrained optimization

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
```