Knowledge-based configuration: Modeling in MiniZinc

Kostyantyn Shchekotykhin

Alpen-Adria-Universität Klagenfurt, Austria

Agenda

Section 1

Constraint satisfaction problems

Constraint satisfaction problems (CSPs)

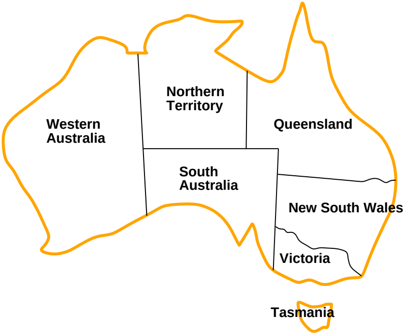
- Standard search problem:
  - **state** is a "black box" – any old data structure that supports goal test, eval, successor
- CSP:
  - **state** is defined by variables X_i with values from domain D_i
  - **goal test** is a set of constraints specifying allowable combinations of values for subsets of variables
- Simple example of a **formal representation language**
- Allows useful **general-purpose** algorithms with more power than standard search algorithms

Vertex-Coloring problem

- Given an undirected graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, and a set of colors C find such an assignment of colors to vertices that for any edge $(v_1, v_2) \in E$ colors of the vertices v_1 and v_2 are different.
- The problem is an abstraction of many practical problems:
  - Configuration of electronic circuits – identification of groups of non-conflicting components
  - Compiler optimization – registry allocation (most often used color)
  - Scheduling – schedule jobs in time slots and avoid conflicts (same color)
  - Pattern matching, e.g. in biochemistry dealing with protein fragments

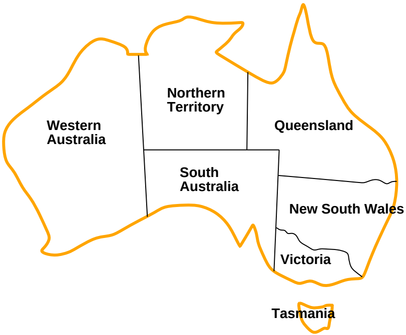
Example: Map-Coloring I

- Map-Coloring is a special case of the graph-coloring problem
- Given a map assign colors to territories in such a way that no two neighboring territories have the same color
- Consider a map of Australia:



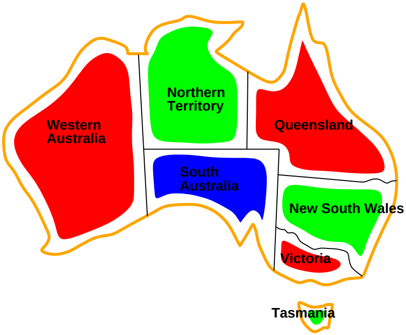
Example: Map-Coloring II

- **Variables** WA, NT, Q, NSW, V, SA, T
- **Domains** $D_i = \{red, green, blue\}$
- **Constraints**: adjacent regions must have different colors e.g., $WA \neq NT$ (if the language allows this), or $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$



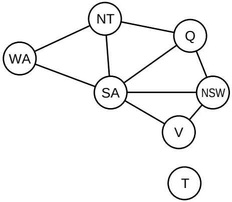
Example: Map-Coloring III

- **Solutions** are assignments satisfying all constraints, e.g., $\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$



Constraint graph

- Binary CSP: each constraint relates at most two variables
- Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search.

Example

Tasmania is an independent subproblem!

Varieties of CSPs

Discrete variables

- finite discrete domains
  - n variables with d possible values $\Longrightarrow$ $O(d^n)$ complete assignments
  - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
- infinite domains (integers, strings, etc.)
  - e.g., job scheduling, variables are start/end days for each job
  - enumeration of possible assignments is not possible
  - specific constraint languages, e.g. $StartJob_1 + 5 \leq StartJob_3$
  - **linear** constraints solvable, **nonlinear** undecidable

Continuous variables

- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in poly time by LP methods

Varieties of constraints

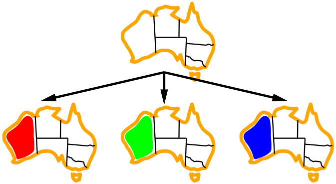
- Unary constraints involve a single variable, e.g., $SA \neq green$
- Binary constraints involve pairs of variables, e.g., $SA \neq WA$
- Higher-order constraints involve 3 or more variables, e.g., cryptarithmetic column constraints
- Preferences (soft constraints), e.g., *red* is better than *green* often representable by a cost for each variable assignment $\rightarrow$ constrained optimization problems

Backtracking search

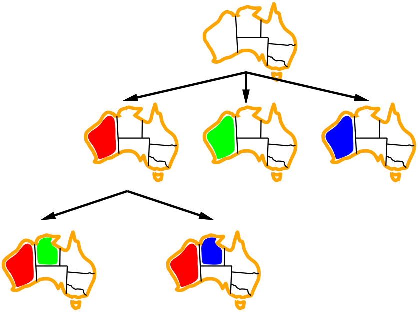
- Variable assignments are commutative, i.e., [$WA = red$ then $NT = green$] same as [$NT = green$ then $WA = red$]
- Only need to consider assignments to a single variable at each node $\implies$ $b = d$ and there are d^n leaves
- Depth-first search for CSPs with single-variable assignments is called backtracking search
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for $n \approx 25$

Backtracking example

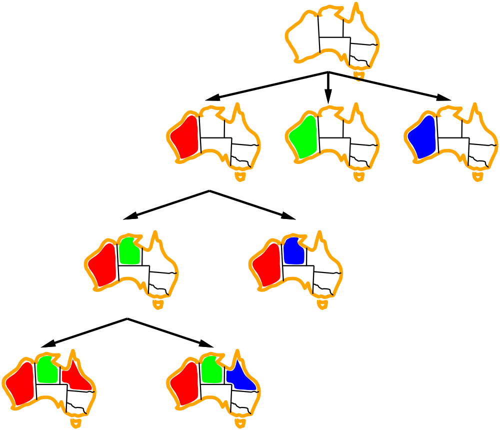
Backtracking example



Backtracking example



Backtracking example



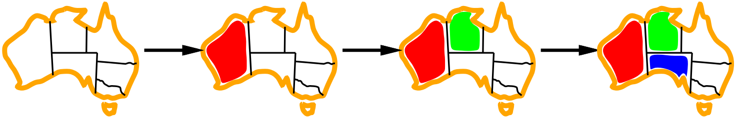
Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

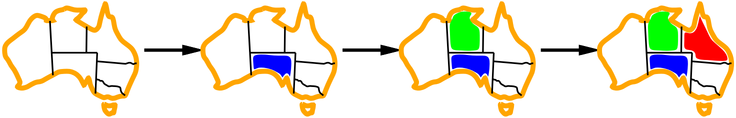
Minimum remaining values

- **Minimum remaining values** (MRV): choose the variable with the fewest legal values



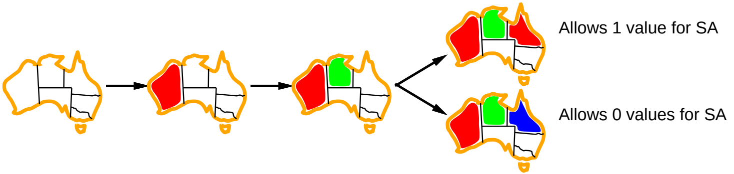
Degree heuristic

- In some cases MRV does not discriminated between the variables, i.e. all variables have domains of equal cardinality.
- **Degree heuristic**: choose the variable with the most constraints on remaining variables



Least constraining value

- We have selected a variable, but which value should we try first?
- Choose the **least constraining value**: the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible!

Forward checking & Arc consistency

Forward checking:

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Arc consistency is the simplest form of propagation which makes each arc consistent

Definition

$X \rightarrow Y$ is consistent iff for **every** value x of X there is **some** allowed y

Local search for CSPs

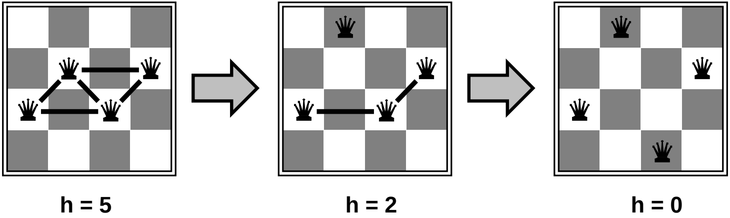
- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators **reassign** variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:

Definition

Choose value that violates the fewest constraints, i.e. hillclimb with $h(n)$ = total number of violated constraints

Example: 4-Queens

- **States**: 4 queens in 4 columns ($4^4 = 256$ states)
- **Operators**: move queen in the column
- **Goal test**: no attacks
- **Evaluation**: $h(n)$ = number of attacks



Summary

- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking is a depth-first search with one variable assigned per node
- Variable and value heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- The CSP representation allows analysis of problem structure
- Tree-structured CSPs can be solved in linear time
- Local search allows to get an approximation of a solution in practice

Section 2

MiniZinc basics

MiniZinc Resources

- Main page
- G12 MiniZinc distribution
  please select the latest version and your platform below
- MiniZinc IDE
- Tutorial
- Language specifications
- Gecode solver for FlatZinc or see MiniZinc Challenge for the others
- Håkan Kjellerstrand's Blog

Creating CSP models

- Constraint programming is usually done in two steps:
  - creation of an **conceptual model**, an abstraction of some real-world problem, and
  - design of a **program** that solves the problem
- This process, in most of the cases, requires some experiments with:
  - different models
  - different solving techniques
  - different heuristics/orderings

Modeling languages

- Programming language used with some constraint-solving library: Choco (Java), Gecode(C++),
- Constraint programming language (Zinc/MiniZinc, ECLiPSe, Minion, Prologs)
- Mathematical modeling language (AMPL[1], OPL[2])
- Domain specific language

These languages vary in:

- the **level** of abstraction from an underlying computer architecture
- **expressiveness**, i.e. which problems can be specified in a language

[1] A Mathematical Programming Language
[2] Optimization Programming Language

MiniZinc

- **MiniZinc** is a modeling language being developed mostly by NICTA (Australia)
- Models can be solved with constraint or MIP solver (not all features are supported by MIP)
- MiniZinc $\subset$ Zinc – a more powerful modeling language

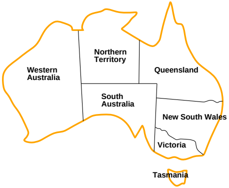
Coloring example modeled in MiniZinc

```
% Colouring Australia using nc colours
int: nc = 3;
var 1..nc: wa; var 1..nc: nt; var 1..nc: sa;
var 1..nc: q; var 1..nc: nsw; var 1..nc: v;
var 1..nc: t;

constraint wa != nt;     constraint wa != sa;
constraint nt != sa;     constraint nt != q;
constraint sa != q;      constraint sa != nsw;
constraint sa != v;      constraint q != nsw;
constraint nsw != v;

solve satisfy;

 output ["wa=", show(wa), "\t nt=", show(nt), "\t sa=", show(sa),
 "\n", "q=", show(q), "\t nsw=", show(nsw), "\t v=", show(v),
 "\n", "t=", show(t), "\n"];
```



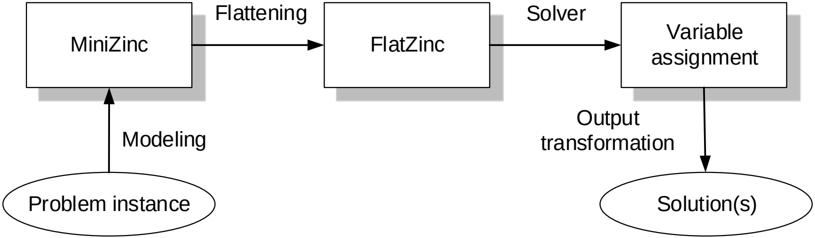
Running MiniZinc

- Save the model in `au.mzn` file and run `mzn au.mzn`
- The output model should look like

```
wa=1     nt=3      sa=2
q=1      nsw=3      v=1
t=1
----------
```

- MiniZinc models must have `mzn` extension and data `dzn`
- use `-f` option to forward the result of "flattening" to other solvers. E.g. `-f fz` will forward fzn file to Gecode FlatZinc front-end
- the output is saved to an ozn file that is then reformatted using the pattern given in output predicate

MiniZinc life-cycle



Parameters and variables I

- **Parameters** correspond to constants in usual programming languages, i.e. they should have a value and only one
- Sample assignments

```
int: i=3;
int: j;    j=3;
```

- Values of **decision variables** are unknown and should be determined by a solver
- Sample variable declarations:

```
var int: x;
var 1..i: y;
```

Parameters and variables II

- Identifiers in MiniZinc start with a letter followed by other letters, underscores or digits
- Moreover, the underscore "_" is the name for an anonymous decision variable
- The basic parameter and variable **types** are:
  - integers **int**; variables also ranges 1..n and sets
  - floating point numbers **float**; variables also ranges 1.0..f and sets
  - booleans **bool**
  - strings **string** (parameters only)
  - Sets: **set of int**: States = 1..7;
  - Arrays: **array**[States] **of var int**: australia;

Operators and expressions

- MiniZinc provides the relational operators: =, !=, <, >, <=, and >=
- Integers: +, -, *, div, mod
- Floats: +, -, *, /
- Casting: **int2float**
- Functions: **abs**, **pow**, (only floats): **sqrt**, **ln**, **sin**, **cos** and others
- Booleans: /\, \/, implications <-, ->, equivalence <->, and negation **not**. Casting **bool2int** allows to convert results of Boolean expressions to integers 0 and 1.

Example: Arithmetic operations I

A backer has 4kg self-raising flour, 6 bananas, 2kg of sugar, 500g of butter and 500g of cocoa and can make two sorts of cakes. A banana cake which takes 250g of self-raising flour, 2 mashed bananas, 75g sugar and 100g of butter, and a chocolate cake which takes 200g of self-raising flour, 75g of cocoa, 150g sugar and 150g of butter. We can sell a chocolate cake for $4.50 and a banana cake for $4.00. The question is how many of each sort of cake should the baker bake to maximize the profit.

Example: Arithmetic operations II

```
int: priceChoco = 450;   int: priceBanana = 400;
var 0..100: b; % no. of banana cakes
var 0..100: c; % no. of chocolate cakes
% flour
constraint 250*b + 200*c <= 4000;
% bananas
constraint 2*b <= 6;
% sugar
constraint 75*b + 150*c <= 2000;
% butter
constraint 100*b + 150*c <= 500;
% cocoa
constraint 75*c <= 500;
% maximize our profit
solve maximize priceBanana*b + priceChoco*c;
output ["no. of banana cakes = ", show(b), "\n",
"no. of chocolate cakes = ", show(c), "\n"];
```

Output

- The output statement **output** is followed by a list of strings that should be used to format the solution
- Strings can be concatenated by the operator ++
  “Northern” ++ “ Territories” = “Northern Territories”
- **show**(X) is used to retrieve values of variable and parameters as srings

Data files

- Models can be used with different input parameters
- MiniZinc allows specification of parameters in separate data files (extension `dzn`)
- In the bakery example different prices for the cakes as well as different amounts of components used in the cakes can be specified separately from the model (`bakery.dzn`)

```
int: priceChoco = 450;    int: priceBanana = 400;
int: flour = 4000;        int: bananas = 6;
int: sugar = 2000;        int: butter =  500;
int: cocoa = 500;
```

Basic structure of a model

- A MiniZinc model consists of a sequence of items
- The model is declarative, that is order of items is not important
- Possible items are:
  - An inclusion item **include** <filename (string)>;
  - An output item **output** <list of string expressions>;
  - Declaration of a variable
  - A constraint **constraint** <Boolean expression>;
  - A solve item (only one of the following is allowed)
    - **solve satisfy**;
    - **solve maximize** <arith. expression>;
    - **solve minimize** <arith. expression>;
  - **predicate** and test (**assert**) items
  - Search annotation items **ann**

Section 3

Arrays and sets

Sets

Sets are declared by

```
set of <type> : <name>
```

- Allowed types: integers, floats or Booleans.
- Set can be declared as $\{e_1, \ldots, e_n\}$
- Generated sequences of integers as *lower..upper*
- Set operations:

  card, **in**, **union**, **intersect**, **subset**, **superset**, **diff**, **symdiff**

  Examples:

```
set of int: Products = {1,2} union {3,4};
int: setCardinality = card(Products);
```

Arrays

- An array is declared by

  array [index_set_1,index_set_2, ...] **of** <type> : <name>

- Index sets of an array are sets of integers
- Elements of an array can be parameters or decision variables
- Built-in function **length** returns the number of elements in a dimension of an array
- Concatenation of two arrays ++

```
array[States] of string: names;
array[States,States] of 0..1: inc;
array[States] of var Colors: aust;
int: len = length(inc) ;
% len = 49 = States * States = 7*7

names = ["wa", "nt", "sa", "q"] ++ ["nsw", "v", "t"];
inc =  [| 0,1,1,0,0,0,0,
        | 1,0,1,1,0,0,0, ... |];
```

Arrays/Sets Comprehensions

- A set comprehension has form

```
{ expr | generator_1, generator_2, ... where <bool-expr> }
```

- An array comprehension is similar

```
[ expr | generator_1, generator_2, ... where <bool-expr> ]
```

```
{i + j | i, j in 1..10 where j < i /\ i < 4} =
                  = {2 + 1, 3 + 1, 3 + 2} = {3, 4, 5}
```

Iteration

MiniZinc provides a variety of built-in functions for iterating over a list or set:

- Numbers: **sum**, **product**, **min**, **max**
- Constraints: **forall**, **exists**

```
forall (i, j in 1..10 where i < j) (a[i] != a[j]);
% is equivalent to
forall ([a[i] != a[j] | i, j in 1..10 where i < j]);
int: maxColor = max(s in States) (aust[s]);
```

A general model of the coloring example I

- Data file containing a problem instance:

```
Colors = 1..3;
States = 1..7;

names = ["wa", "nt", "sa", "q", "nsw", "v", "t"];

inc =     [| 0,1,1,0,0,0,0,
           | 1,0,1,1,0,0,0,
           | 1,1,0,1,1,1,0,
           | 0,1,1,0,1,0,0,
           | 0,0,1,1,0,1,0,
           | 0,0,1,0,1,0,0,
           | 0,0,0,0,0,0,0 |];
```

A general model of the coloring example II

- Model of the coloring problem:

```
set of int: Colors;
set of int: States;
array[States] of string: names;
% incidence matrix for the states
array[States,States] of 0..1: inc;

array[States] of var Colors: aust;

constraint forall (st1, st2 in States where inc[st1,st2] > 0) (
  aust[st1] != aust[st2]
);

solve satisfy;
output [names[state] ++ "=" ++ show(aust[state]) ++ "\t" | state
     in States];
```

Section 4

Advanced modeling

Global constraints

- MiniZinc has a big library of efficiently implemented global constraints[3]

```
include "globals.mzn";
alldifferent(array[int] of var int:x)
table(array[int] of var int: x, array[int,int] of int:t)
global_cardinality(array[int] of var int: x,
                   array[int] of int: cover,
                   array[int] of var int: counts)
```

- `alldifferent` all variables in the array x must take different values.
- `table` constrains values of variables in x to the ones given in the table t (a variable per row)
- `global_cardinality` requires that the number of occurrences of $cover[i]$ in x is $counts[i]$.

[3] See http://www.minizinc.org/downloads/doc-1.6/mzn-globals.html

Local Variables

- It is often useful to introduce local variables in a test or predicate
- The `let` expression allows you to do so

```
let { <var_dec>, ...} in <exp>
```

- It can also be used in other expressions
- The variable declaration can contain decision variables and parameters
- Parameters must be initialized

```
constraint let { var int: s = x1 + x2 + x3 + x4,
                 int l = lb(x1), int u = ub(x4) } in
           l <= s /\ s <= u;
```

Efficiency in MiniZinc

- A problem can be modeled in many different ways
- Not every implementation can be solved efficiently
- Information about efficiency is obtained using the MiniZinc flags
  - solver-statistics [number of choice points]
  - statistics [number of choice points, memory and time usage]
- Extensive experimentation is required to determine relative efficiency

Writing Efficient Models I

- Add **search annotations** to the solve item to control exploration of the search space

```
<search_type>(variables, varchoice, constrainchoice,
  strategy)
solve :: int_search(q, first_fail, indomain_min, complete)
satisfy;
```

  - Types: `int_search`, `bool_search` (arrays), `set_search`
  - Choose the variable:
    - `input_order` in order from the array,
    - `first_fail` (MRV) with the smallest domain size,
    - `most_constrained` with the smallest domain, breaking ties using the number of constraints
  - Values: `indomain_min` assign the smallest domain value[4]

Writing Efficient Models II

- Use global constraints such as `alldifferent` since they have better propagation behavior
- Try different models for the problem
- Add redundant constraints
- Bound variables as tightly as possible (avoid **var int**)
- Avoid introducing unnecessary variables

Expert users:

- Extend the constraint solver to provide a problem specific global constraint
- Extend the constraint solver to provide a problem specific search routine

[4] See http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf Section 5.6.1 for a complete list of heuristics

Section 5

House configuration problem

House problem

Customer requirements

1. Declaration of available persons, things and ownership relations between them

Configuration requirements

1. each thing must be stored in exactly one cabinet
2. a cabinet can contain at most 5 things
3. every cabinet must be placed in exactly one room
4. a room can contain at most 4 cabinets
5. each room belongs to a person
6. and a room may only contain cabinets storing things of the owner of the room

Goal store all things in a house such that the set of requirements is fulfilled

MiniZinc encoding I

```
include "globals.mzn";
% input
int: cabinetCap = 5; % capacity of a cabinet
int: roomCap = 4;    % capacity of a room

array [Things] of Persons : p2t;

set of int: PersonsDomain;
set of int: ThingsDomain;
set of int: CabinetsDomain;
set of int: RoomsDomain;

% enumeration of set elements
set of int: Persons = 1..card(PersonsDomain);
set of int: Things = 1..card(ThingsDomain);
set of int: Cabinets = 1..card(CabinetsDomain);
set of int: Rooms = 1..card(RoomsDomain);
```

MiniZinc encoding II

```
% an array of length |Things| of variables with domain Cabinets
array [Things] of var Cabinets : t2c;
array [Things] of var Rooms : t2r;
array [Cabinets] of var Rooms : c2r;

array[Cabinets] of int : cabinetLower = [0 | i in Cabinets];
array[Cabinets] of int : cabinetUpper =
                              [cabinetCap | i in Cabinets];

% Built-in global constraint. Each value should belong to the
    set Cabinets (i.e. closed) converted to an array, and each
    element of Cabinets can be used at least cabinetLower[i]
    and at most cabinetUpper[i] times
constraint global_cardinality_low_up_closed(t2c,
        [i | i in Cabinets], cabinetLower, cabinetUpper);
```

MiniZinc encoding III

```
% capacity contraint for rooms
array[Rooms] of int : roomLower = [0 | i in Rooms];
array[Rooms] of int : roomUpper =
                              [roomCap*cabinetCap | i in Rooms];

% each room can contain at most 20 things
constraint global_cardinality_low_up_closed(t2r, [i | i in Rooms],
     roomLower,  roomUpper);

array[Rooms] of int : roomCabinetUpper = [roomCap | i in Rooms];
constraint global_cardinality_low_up_closed(c2r, [i | i in Rooms],
     roomLower,  roomCabinetUpper);
```

MiniZinc encoding IV

```
% if thing is stored in a cabinet and in a room then the cabinet
    is placed in the room
constraint forall (i in Things) (
 let {var int: cabinet = t2c[i], var int: room = t2r[i]} in
    c2r[cabinet] = room);

% if 2 different things are placed in the same cabinet they have
    to be owned by the same person
constraint forall (i,j in Things where i != j /\ p2t[i] != p2t[j]) (
t2c[i] != t2c[j]);

% if 2 different things are placed in the same room they have to
    be owned by the same person
constraint forall (i,j in Things where i != j /\ p2t[i] != p2t[j]) (
t2r[i] != t2r[j]);
```

MiniZinc encoding V

```
ann: search_ann;
solve
:: search_ann
satisfy;

output ["t2c(" ++ show(ThingsDomain[i]) ++ "," ++
    show(CabinetsDomain[t2c[i]]) ++ "). " | i in Things]
++ ["c2r(" ++ show(CabinetsDomain[t2c[i]]) ++ "," ++
    show(RoomsDomain[t2r[i]]) ++ "). " | i in Things]
```

Data file

This instance presented in the first lecture

```
% input relations
p2t = [1,1,1,1,1,2];

%sets of names
CabinetsDomain = {500,501,502,503,504};
RoomsDomain = {1000,1001,1002,1003,1004};
PersonsDomain = {1,2};
ThingsDomain = {3,4,5,6,7,8};

% search annotation
search_ann = int_search(t2c, first_fail, indomain_max,
    complete));
```

Optimization

```
% returns true if a value occurs in the array
predicate used(array[int] of var int: a, int: b) =
      exists (t in index_set(a)) (a[t]==b);

% costs defined in the data file
int: roomCost;
int: cabinetCost;

% computation of costs of an assignment
var int: costs =
      sum (r in Rooms)(roomCost*bool2int(used(t2r,r)))
    + sum (c in Cabinets)(cabinetCost*bool2int(used(t2c,c)));

solve :: search_ann
    minimize costs;
```

Summary

General observations:

- Configuration problems can be modeled in MiniZinc
- Solution of the model can be found using any FlatZinc solver
- Various heuristics and orderings can improve the performance
  - `input_order` heuristic allows to specify any cariable selection order
- Readability of the language is average

Source files for MiniZinc 1.6[5]

- Model house.mzn
- Data house.dzn

[5] Click on the file names to get the model and data file attached to pdf