Topic 3: Constraint Predicates [1]
(Version of 26th September 2025)

Pierre Flener, Gustav Björdal,
and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL451:
Modelling for Combinatorial Optimisation

[1] Many thanks to Guido Tack for feedback

Outline

UPPSALA UNIVERSITET

Outline

Examples

Let `A` be an array of decision variables:

- The `all_different(A)` constraint holds if and only if all the elements of `A` take distinct values:

```
forall(i,j in index_set(A) where i < j)(A[i] != A[j])
```

- The `count(A,v) >= c` constraint holds if and only if the number of occurrences in `A` of `v` is at least `c`, where `v` and `c` can be decision variables:

```
sum(x in A)(x = v) >= c
```

Definition

A definition of a constraint predicate is its meaning,
stated in MiniZinc in terms of usually simpler constraint predicates.

Examples

See some MiniZinc-provided default definitions at slide 4.

Definition

Each use of a predicate is decomposed during flattening by inlining
either its MiniZinc-provided default definition
or an overriding backend-provided solver-specific definition.

Examples

If a predicate γ on arguments X is supported by a solver,
then its backend provides $\gamma(X) = \gamma(X)$ as solver-specific definition.

Motivation:

+ More compact and intuitive models,
  because more expressive predicates are available: islands of common combinatorial structure are identified in declarative high-level abstractions.

+ Faster solving, due to better reasoning and relaxation,
  enabled by more global information in the model,
  provided the predicate is a built-in of the used solver.

Enabling constraint-based modelling:

- Constraint predicates over any number of decision variables go by many names: global-constraint predicates, combinatorial predicates, ...
- See the MiniZinc global constraints and the Global-Constraint Catalogue.
- Some predicates cannot be reified, say via `bool2int`.

Outline

UPPSALA UNIVERSITET

Definition (Laurière, 1978)

The `all_different(X)` constraint holds if and only if all the elements of the array `X` of decision variables take distinct values.

Its default definition is a conjunction of $\frac{n \cdot (n-1)}{2}$ disequality constraints when `X` has `n` elements:

```
forall(i,j in index_set(X) where i < j)(X[i] != X[j])
```

The `all_different_except(X,S)` constraint allows multiple occurrences of the exception values in the set `S`.

Examples

- *n*-Queens problem: see Topic 1: Introduction.
- Photo Alignment problem: see Topic 2: Basic Modelling.
- Student Seating problem: see Topic 4: Modelling.
- Object, Shapes, and Colours: see Topic 4: Modelling.

Outline

Definition (Pachet and Roy, 1999)

The `nvalue(m,X)` constraint holds if and only if decision variable `m` takes the number of distinct values taken by the elements of the array `X` of decision variables. If array `X` is 1d and has indices `1..n`, then this means:

$$|\{\texttt{X[1]},\ldots,\texttt{X[n]}\}| = \texttt{m}$$

The expression `nvalue(X)` denotes the number of distinct values taken by the elements of the array `X` of decision variables.

If $|\texttt{X}| = \texttt{n}$ then `nvalue(n,X)` means `all_different(X)`, but:
Always use the most specific available constraint predicate!

Example

Model 2 of the Warehouse Location problem: see Topic 6: Case Studies.

Outline

Definition (Régin, 1996)

The `global_cardinality(X,V,C)` constraint holds if and only if each decision variable `C[j]` takes the number of elements of the array `X` of decision variables that take the given *value* `V[j]`. Variant predicates exist.
Add `_closed` to the predicate name if `V` is the domain of the variables in `X`.

Its default definition in MiniZinc includes:

```
forall(j in index_set(V))(count(X,V[j]) = C[j])
```

It means `all_different(X)` if $V = \bigcup_i \mathrm{dom}(X[i])$ and $\mathrm{dom}(C[j]) = \{0, 1\}$ for each `j`, but: Always use the most specific available constraint predicate!

Examples

- Magic Series problem + Student Seating problem + Object, Shapes, and Colours: see Topic 4: Modelling.
- Warehouse Location + Sports Scheduling: see Topic 6: Case Studies.

A Common Source of Inefficiency in Models

Example

The model snippet

```
constraint forall(j in index_set(V))
                 (count(X,V[j]) = C[j]);
```

should be reformulated, due to the shared array `X` for each `j`, into:

```
constraint global_cardinality(X,V,C);
```

by applying the default definition backwards:

- at worst, it will be applied forwards while flattening;
- at best, the invoked solver has better reasoning.

This advice holds for each global-constraint predicate,
and for all (quantified) constraints over *shared* decision variables.

Outline

Definition (Van Hentenryck and Carillon, 1988)

The `element(i,X,e)` constraint, where:

- `X` is an array of decision variables,
- `i` is an integer decision variable, and
- `e` is a decision variable,

holds if and only if `X[i] = e`.

For better model readability,
the `element` predicate should not be used,
as the functional form `X[`ϕ`]` is allowed,
even when ϕ is an integer expression involving at least one decision variable.

Use: The `element` predicate and its functional form `X[`ϕ`]` help model an unknown element of an array.

Example (Job allocation at minimal salary cost)

Given jobs `Jobs` and the salaries of work applicants `Apps`, **find** a work applicant for each job **such that** some constraints (on the qualifications of the work applicants for the jobs, on workload distribution, etc) are satisfied and the total salary cost is minimal:

```
1 array[Apps] of 0..1000: Salary;  % Salary[a] = cost per job to appl. a
2 array[Jobs] of var Apps: Worker; % Worker[j] = appl. allocated job j
3 solve minimize sum(j in Jobs)(Salary[Worker[j]]);
4 constraint ...;          % qualifications, workload, etc
```

Line 3 is equivalent to the less readable formulation, and flattened into it:

```
array[Jobs] of var 0..max(Salary): Cost; % Cost[j] = salary for job j
constraint forall(j in Jobs)(element(Worker[j],Salary,Cost[j]));
solve minimize sum(Cost);
```

We do not know at modelling time the worker allocated to each job!

Outline

Definition

Let item `i` have the given (weight or) volume `Volume[i]`.
Let decision variable `Bin[i]` denote the bin into which item `i` is put.
Let decision variable `Load[b]` denote the load (= volume of items) of bin `b`.
The `bin_packing_load(Load,Bin,Volume)` constraint holds if and only if each `Load[b]` is the sum of the `Volume[i]` where `Bin[i]` equals `b`.
Variant predicates exist (such as `bin_packing` in the following example).

Example (Balanced academic curriculum problem)

Given, for each course `c` in `Courses`, a workload `W[c]` and a set `Pre[c]` of prerequisite courses, **find** a semester `Sem[c]` in `1..n` for each course `c` in order to satisfy all the course prerequisites under a balanced workload:

```
constraint bin_packing(sum(W) div n, Sem, W); % same load
constraint forall(c in Courses, p in Pre[c])
  (Sem[p] < Sem[c]);
```

A Common Source of Inefficiency in Models

Example

The model snippet

```
constraint forall(b in Bins)
  (Load[b] = sum(i in Items where Bin[i] = b)(Vol[i]));
```

should be reformulated — due to the shared array `Bin` for each `b` and due to the `where` clause on the decision variables `Bin[i]` — as follows:

```
constraint bin_packing_load(Load,Bin,Vol);
```

There are many incarnations of this pattern:

- `Bins` = semesters; `Items` = courses; `Bin[i]` = semester of course `i`; `Vol[i]` = credits for course `i`; `Load[b]` = credits for courses in sem. `b`;
- `Bins` = staff; `Items` = tasks; `Bin[i]` = employee assigned to task `i`; `Vol[i]` = reward for task `i`; `Load[b]` = income over tasks to employee `b`.

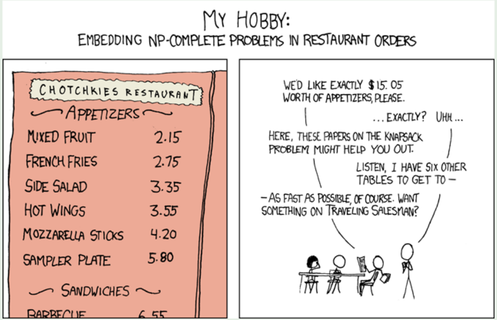
Definition

Let item type `t` have the given (weight or) volume `Volume[t]`.
Let item type `t` have the given (value or) profit `Profit[t]`.
Let decision variable `X[t]` denote the number of items of type `t` that are put into a given knapsack.
Let decision variable `v` denote the total volume of what is in the knapsack.
Let decision variable `p` denote the total profit of what is in the knapsack.
The `knapsack(Volume,Profit,X,v,p)` constraint holds if and only if both `sum(t in index_set(X))(Volume[t] * X[t]) = v` and `sum(t in index_set(X))(Profit[t] * X[t]) = p` hold.

Example

To model the Knapsack Problem for a knapsack of given capacity `c`, add `constraint v <= c` and state `solve maximize p`.

Example (https://xkcd.com/287)



A simplified version of the Knapsack Problem, but still NP-hard (see an interview for some interesting trivia).

```
enum  Appetisers = {fruit,fries,salad,hotWings,mozzSticks,sampler};
array[Appetisers] of int: Cost = [215,275,335,355,420,580];
array[Appetisers] of int: Joy  = [  0,  0,  0,  0,  0,  0];
array[Appetisers] of var 0..(1505 div min(Cost)): Amount;
constraint knapsack(Cost, Joy, Amount, 1505, 0);
solve satisfy;
```

Outline

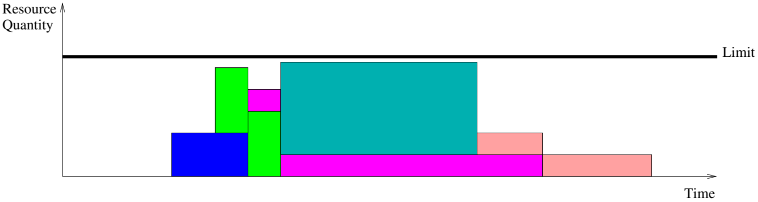
Assume we need to schedule a set of non-interruptible tasks under constraints (on resources, precedences, ...) such that the last task has the earliest end.

Definition

A task $T_{\mathtt{i}}$ is a triple $\langle$`S[i]`,`D[i]`,`R[i]`$\rangle$ of parameters or variables, where:

- `S[i]` is the starting time of task $T_{\mathtt{i}}$
- `D[i]` is the duration of task $T_{\mathtt{i}}$
- `R[i]` is the quantity of a global reusable resource needed by $T_{\mathtt{i}}$

Tasks may be run in parallel when the capacity of the global resource suffices.

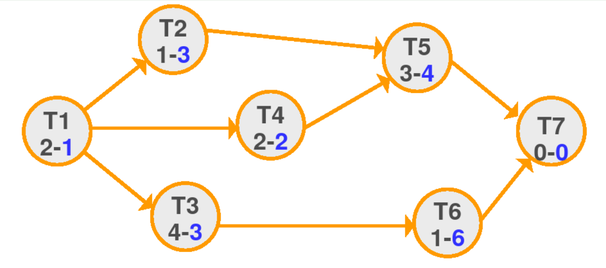


Schedule with parallel tasks and a capacitated global reusable resource

Definition

A precedence constraint of task T_1 on task T_2 requires that T_1 ends before or when T_2 starts. We say that task T_1 precedes task T_2.

Example (courtesy Magnus Rattfeldt)



Sample tasks (circles), durations (black numbers), resource requirements (blue numbers), and precedences (orange arrows). Task T7 is a dummy task, as we do not know which of tasks T5 and T6 will end last.

Let us temporarily ignore the capacitated global reusable resource:
If we have an uncapacitated global reusable resource or each task has enough of its own local reusable resource, then the polynomial-time-solvable problem of finding the earliest ending time, under only the precedence constraints, for performing all the tasks can be modelled using linear inequalities.

Example (continued)

The precedence constraints indicated by the orange arrows on slide 24 are modelled as follows, based on the task durations indicated there in black:

```
constraint D = [2,1,4,2,3,1,0];
constraint S[1]+D[1] <= S[2] /\ S[1]+D[1] <= S[3]
        /\ S[1]+D[1] <= S[4] /\ S[2]+D[2] <= S[5]
        /\ S[3]+D[3] <= S[6] /\ S[4]+D[4] <= S[5]
        /\ S[5]+D[5] <= S[7] /\ S[6]+D[6] <= S[7];
% plug in here the resource constraint of the next slide
solve minimize S[7];
```

Definition (Aggoun and Beldiceanu, 1993)

The `cumulative(S,D,R,c)` constraint, where each task $T_{\mathtt{i}}$ has the starting time `S[i]`, duration `D[i]`, and resource requirement `R[i]`, holds if and only if the resource capacity `c` is never exceeded when performing the $T_{\mathtt{i}}$.

Note that `cumulative` does not ensure any precedence constraints between the tasks: these have to be stated separately (as on the previous slide).

Example (end)

To ensure that the global reusable resource capacity of `c = 8` units, say, is never exceeded under the resource requirements of the tasks indicated in blue on slide 24, plug the following constraint into the model of the previous slide:

```
constraint cumulative(S,D,[1,3,3,2,4,6,0],8);
```

Definition

A non-overlap constraint between tasks T_1 and T_2 requires that either T_1 precedes T_2 or T_2 precedes T_1 (say because both tasks require a resource that is available only for one task at a time).
We say that tasks T_1 and T_2 do not overlap in time.

Definition (Carlier, 1982)

The `disjunctive(S,D)` constraint, where each task `T_i` has the starting time `S[i]` and duration `D[i]`, holds if and only if no two tasks T_i and T_j overlap in time. It is also known as `unary`.

It has among others the following definitions:

- ```
  forall(i,j in 1..n where i<j)
    ((S[i] + D[i] <= S[j]) \/ (S[j] + D[j] <= S[i]))
  ```
- `cumulative(S, D, [1 | i in 1..n], 1)`

Always use the most specific available constraint predicate!
  ```

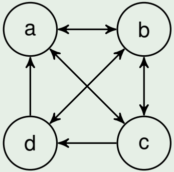
UPPSALA UNIVERSITET

Outline

Enabling the representation of a circuit in a digraph:

- Let decision variable `S[v]` denote the successor of vertex `v` in the circuit.
- The domain of `S[v]` is the set of vertices to which there is an arc from vertex `v`, plus `v` itself (for a reason that will become apparent below).

Example




```
enum Vertices = {a,b,c,d};
array[Vertices] of var Vertices: S;
constraint S[a] != d /\ S[d] != c;
```

Assume the decision variables in `S` take the following values:

- `[b,c,d,a]`: one circuit a → b → c → d → a
- `[c,a,b,d]`: one subcircuit a → c → b → a and `S[d]=d`
- `[a,b,c,d]`: one empty subcircuit: `S[v]=v` for all `v` in `Vertices`
- `[c,d,a,b]`: two subcircuits, namely a → c → a and b → d → b
- `[b,d,a,d]`: c → a → b → d is not a (sub)circuit

Definition (Laurière'78; Beldiceanu and Contejean'94)

The `circuit(S)` constraint holds if and only if the arcs $\mathtt{v} \rightarrow \mathtt{S[v]}$ for all `v` form a Hamiltonian circuit: each vertex is visited exactly once.
The `subcircuit(S)` constraint holds if and only if `circuit(S′)` holds for exactly one possibly empty but non-singleton subarray `S′` of `S`, and $\mathtt{S[v]} = \mathtt{v}$ for all the other vertices `v`.

Examples (Vehicle routing)

Travelling salesperson problem (generalise this for vehicle routing problems with multiple vehicles or with side constraints):

```
solve minimize sum(c in Cities)(Distance[c,Next[c]]);
constraint circuit(Next);
```

Requiring a directed path from vertex `v` to vertex `w`:

```
  constraint subcircuit(S) /\ S[w] = v;
```

upon adding `v` to the domain of `S[w]` if need be.

Many graph constraints, including `dpath`, exist in MiniZinc.

Outline

Example

We have `lex_lesseq([1,2,34,5,678], [1,2,36,45,78])`, because `34 < 36`, even though `678` $\not\leq$ `78`.

Definition

The `lex_lesseq(X,Y)` constraint, where `X` and `Y` are same-length 1d arrays of decision variables, say both with indices in `1..n`, holds if and only if `X` is lexicographically at most equal to `Y`:

- either `n = 0`,
- or `X[1] < Y[1]`,
- or `X[1] = Y[1]` & `lex_lesseq(X[2..n],Y[2..n])`.

Variant predicates exist.

Usage: Exploit index symmetries in matrix models, where there are arrays of decision variables: see Topic 4: Modelling, and see Topic 5: Symmetry.

Outline

Regular Expressions

Examples (Regular Expressions)

- $(\mathbf{0}|\mathbf{1})^*\mathbf{0}$ denotes the set of even binary numbers.
- $\mathbf{1}^*(\mathbf{011}^*)^*(\mathbf{0}|\epsilon)$ denotes the set of strings of zeros and ones without consecutive zeros.
- $(\mathbf{0}|\mathbf{1})^*\mathbf{00}(\mathbf{0}|\mathbf{1})^*$ denotes the set of strings of zeros and ones with consecutive zeros.

Notation for strings:

- Let ϵ denote the empty string.
- Let $v \cdot w$ denote the concatenation of strings v and w.
- Let w^i denote the concatenation of i copies of string w.

UPPSALA UNIVERSITET

Motivation

all_different

nvalue

global_cardinality

element

bin_packing, knapsack

cumulative, disjunctive

circuit, subcircuit

lex_lesseq

regular, table

Checklist

M4CO topic 3

Regular Expressions and Languages

Definition

Let Σ be an alphabet, that is a finite set of symbols. A regular expression r over Σ, and its regular language over Σ, denoted $\mathcal{L}(r)$, are defined as follows:

- $\varnothing$ is a regular expression: $\mathcal{L}(\varnothing) = \varnothing$.
- ϵ is a regular expression: $\mathcal{L}(\epsilon) = \{\epsilon\}$.
- If $\sigma \in \Sigma$, then σ is a regular expression: $\mathcal{L}(\sigma) = \{\sigma\}$.
- If r and s are regular expressions, then rs is a regular expression: $\mathcal{L}(rs) = \{v \cdot w \mid v \in \mathcal{L}(r) \wedge w \in \mathcal{L}(s)\}$.
- If r and s are regular expressions, then $r|s$ is a regular expression: $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.
- If r is a regular expression, then r^* is a regular expression: $\mathcal{L}(r^*) = \left\{w^i \mid i \in \mathbb{N} \wedge w \in \mathcal{L}(r)\right\}$.

Regular Expressions

Common abbreviations for regular expressions:

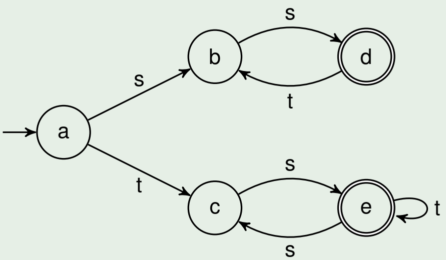
Let r be a regular expression:

- $r^?$ denotes $r|\epsilon$; example in MiniZinc syntax: `"12?"`
- r^+ denotes rr^*; example in MiniZinc syntax: `"34+"`
- r^4 denotes $rrrr$; example in MiniZinc syntax: `"56{4}"`
- [**1 2 3 4**] denotes **1**|**2**|**3**|**4**; same syntax in MiniZinc
- [**5-8**] denotes [**5 6 7 8**]; same syntax in MiniZinc
- [**9-11 14**] denotes [**9 10 11 14**]; same syntax in MiniZinc
- . . . (see the MiniZinc documentation)

Usage: Regular expressions are good for the specification of regular languages, but not so good for reasoning on them, where one often uses finite automata instead.

Deterministic Finite Automaton (DFA), Nondet...FA (NFA)

Example (DFA for regular expression $\mathrm{ss(ts)^*|ts(t|ss)^*}$ over $\Sigma = \{s, t\}$)



Conventions:

- Start state, marked by an arc coming in from nowhere: a.
- Accepting states, marked by double circles: d and e.
- Determinism: exactly one outgoing arc per $\sigma \in \Sigma$. Convention: non-drawn arcs go to a non-accepting missing state with self-loops on each $\sigma \in \Sigma$.

Definition (Pesant, 2004)

The `regular(X,T,q0,A)` constraint holds if and only if the values of the 1d array `X` of decision variables form a string of the regular language accepted by the DFA with alphabet Σ, states Q, transition function $\mathtt{T}\colon Q \times \Sigma \to Q$, start state $\mathtt{q0} \in Q$, accepting states $\mathtt{A} \subseteq Q$. Variants exist, including `regular_nfa`. The `regular(X,r)` constraint holds if and only if the values of `X` form a string of the regular language denoted by the regular expression `r`.

Example (↗)

```
enum Alphabet = {s,t};  enum State = {a,b,c,d,e};
array[State,Alphabet] of opt State:
  Transition = [| b,c | d,<> | e,<> | <>,b | c,e |];
array[1..n] of var Alphabet: X;
constraint regular(X,Transition,a,{d,e});
constraint regular(X,"s s (t s)* | t s (t | s s)*");
```

Definition

The `table(X,T)` constraint holds if and only if the values of the 1d array `X` of decision variables form a row of the 2d array `T` of values.

The 2d array `T` gives an extensional definition of a new constraint predicate, as opposed to the intensional definition so far for all other constraint predicates. Note that `regular` and its variants are *intensional* as an automaton or regular expression is *independent* of the length of `X`.

Example (↗)

If the array `X` of the `regular` constraint of the previous slide for the DFA of two slides ago has `n=4` decision variables, then that constraint is equivalent to:

```
constraint table(X,[| s,s,t,s | t,s,s,s | t,s,t,t |]);
```

Example (The Nonogram Puzzle: instance)

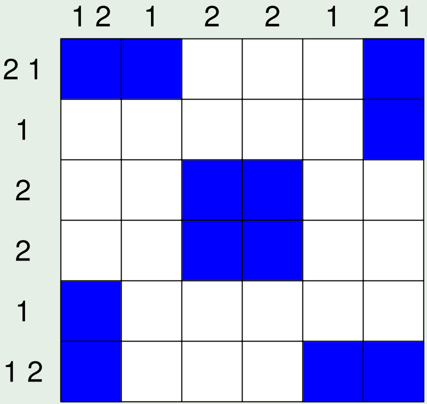
Each hint gives the sequence of lengths of blue blocks in its row or column, with at least one white cell between blocks, but possibly none before the first block or after the last block (or both).

	1 2	1	2	2	1	2 1
2 1						
1						
2						
2						
1						
1 2						

Example (The Nonogram Puzzle: instance)

Each hint gives the sequence of lengths of blue blocks in its row or column, with at least one white cell between blocks, but possibly none before the first block or after the last block (or both).

Solution:



Example (The Nonogram Puzzle: model + data)

Model:

- Decision variables: An enumeration-type decision variable for each cell, with value `w` if it is coloured white, and value `b` if it is coloured blue.
- Constraints: State a `regular` constraint for each hint. For example, for a hint 2 3 1 on a row or column `X` of length $n \geq 8$, state the constraint `regular(X, "w* b{2} w+ b{3} w+ b{1} w*")`.

See Survey of Paint-by-Number Puzzle Solvers: the straightforward model outlined above fares well, at least with a CP solver, compared to programs.

Example (Nurse Rostering)

Each nurse is assigned each day to one of the following:

- `r` regular shift (this value is not available on Sundays)
- `e` extended shift (this value is not available on Sundays)
- `s` Sunday shift (this value is only available on Sundays)
- `o` day off

The labour union of the nurses imposes the following regulations:

- Monday off after a Sunday shift
- No single extended shifts
- One day off after two consecutive extended shifts

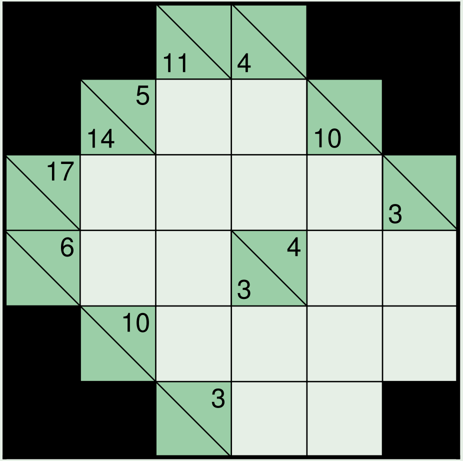
For each nurse `n`, state the following constraint over the scheduling horizon, starting on a Sunday (and typically 17 weeks longs in Sweden):

```
regular(Roster[n,..], "(s o | e e o | r | o)*")
```

Further, a hospital has constraints on nurse presence, on the `Roster[..,d]`.

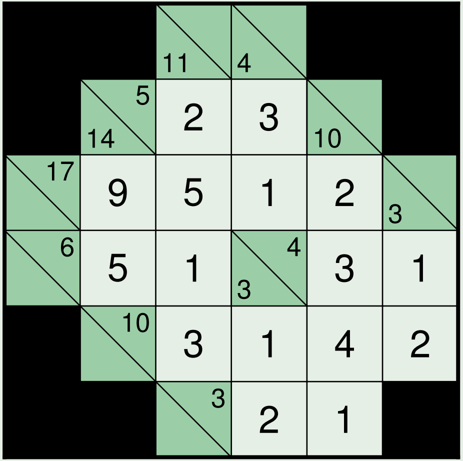
Example (The Kakuro Puzzle: instance)

Fill in digits of `1..9` such that the digits of each word are distinct and add up to the sum to the left (for horizontal words) or top (for vertical words) of the word.



Example (The Kakuro Puzzle: instance)

Fill in digits of `1..9` such that the digits of each word are distinct and add up to the sum to the left (for horizontal words) or top (for vertical words) of the word.



Example (The Kakuro Puzzle: first model)

Model:

- Decision variables: A decision variable for each cell, with domain `1..9`.
- Constraints: For each row or column hint $\texttt{K}[\alpha] + \cdots + \texttt{K}[\beta] = \sigma$, state `all_different(i in` $\alpha..\beta$`)(K[i])`
  `/\ sum(i in` $\alpha..\beta$`)(K[i]) =` σ.

Performance, using a CP solver:

- 22×14 Kakuro with 114 hints: 9,638 nodes, 160 s
- 90×124 Kakuro with 4,558 hints: ? nodes, ? years

Symptom: The definition as two constraints may give weak reasoning: for `x!=y /\ x+y=4`, CP reasoning gives `x,y in 1..3`, not noticing that `2` should be pruned from both domains. We want a custom predicate `all_different_sum`, constraining up to 9 variables over the domain `1..9`.

Example (The Kakuro Puzzle: second model)

New model: Use the `regular` or `table` predicate for the conjunction of the `all_different` and `sum`-based constraints of each hint?

- For each length-2 hint `x+y=4`, state `regular([x,y],"1 3|3 1")`. Note that we have *precomputed* that `y != 2` for this particular case of the wanted `all_different_sum`(X,σ), where $X =$ `[x,y]` and $\sigma = 2$.
- For each length-2 hint `y+z=3`, state `regular([y,z],"1 2|2 1")`.
- One can also use `table` instead:
  `table([x,y],[|1,3|3,1|]) /\ table([y,z],[|1,2|2,1|])`.
- The regular expressions and tables above are not derived parameters, but precomputed solution sets to islands of common combinatorial structure within all Kakuro puzzles. We revisit precomputation in Topic 4: Modelling.
- But what about the length-9 hint $\texttt{K}[\alpha] + \cdots + \texttt{K}[\alpha+8] = 45$? There are $9! = 362{,}880$ solutions to this hint...

Example (The Kakuro Puzzle: second model, end)

New model (end):

- For each length-9 hint `K[`α`] + ··· + K[`α`+8] = 45`, it suffices to state `all_different([K[i] | i in `α`..`α`+8]`, as the sum of 9 distinct non-0 digits is necessarily 45.
- For each length-8 hint `K[`α`] + ··· + K[`α`+7] = `σ, it suffices to state `all_different([K[i] | i in `α`..`α`+7] ++ [45-`σ`])`.
- For each hint `K[`α`] = `σ, it suffices to state `K[`α`] = `σ.

Other opportunities for improvement exist.

New performance, using a CP solver:

- 22×14 Kakuro with 114 hints: 0 search nodes, 28 ms!
- 90×124 Kakuro with 4,558 hints: 0 nodes, 345 ms!

Published diabolically hard Kakuros (like the 22×14 one mentioned above) where the new model pays off are rare.

When to Use These Predicates?

Rapid prototyping of a new constraint predicate: The `regular` and `table` predicates are very useful in the following conjunctive situation:

- A needed constraint predicate γ on a 1d array of decision variables is not a built-in of MiniZinc or the used solver.
- A definition of γ in terms of built-in predicates is not apparent to the modeller, or such a definition has turned out to inherit reasoning that either has too high time complexity or is too weak (or both).
- The modeller does not have the time or skill to design a reasoning algorithm for γ, or deems γ not reusable for other problems.
- The time complexity and strength of a reasoning algorithm for γ are not deemed crucial for the time being.

Important Modelling Idea

Example (Encoding a function on a small set)

The non-linear constraint `x*x=y`, where there is exactly one `y` for every `x`, may yield poor reasoning and become a bottleneck: for `x` only in `1..9`, say, try `element(x, [d*d | d in 1..9], y)`, where `d*d` is not non-linear, that is `[d*d | d in 1..9][x] = y`, for better reasoning and higher speed.

The `element` predicate is a specialisation of `regular` and `table`, just like a function is a special case of a relation:

Example (Encoding a relation over a small set)

The non-linear constraint `x*x=abs(y)`, where there are two `y` for most `x`, may yield poor reasoning and become a bottleneck: for `x` only in `0..3`, say, try the less readable `table([x,y], [|0,0|1,-1|1,1|2,-4|2,4|3,-9|3,9|])` for better reasoning and higher speed (but maybe not with a MIP solver).

Bibliography

Pesant, Gilles.
A regular language membership constraint for finite sequences of variables.
Proceedings of CP 2004, Lecture Notes in Computer Science 3258, pages 482 – 495. Springer, 2004.

Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D.
Introduction to Automata Theory, Languages, and Computation.
Third edition. Addison-Wesley, 2007.

Outline

Checklist for Designing or Reading a Model

12 Predicates with the most specific meanings are used [Ex1, Ex2, Ex3]

13 Global constraints are used, instead of their definitions [Ex1, Ex2]

14 Constraints over shared decision variables are ideally merged [Ex1, Ex2]

15 The `element` predicate is not used explicitly, for readability [Ex]

16 Functions on small sets are encoded by implicit `element`, if need be [Ex]

17 Relations over small sets are encoded by `regular` or `table`, if faster than a formulation in the scope of checklist items 6 to 11 of Topic 2 [Ex]