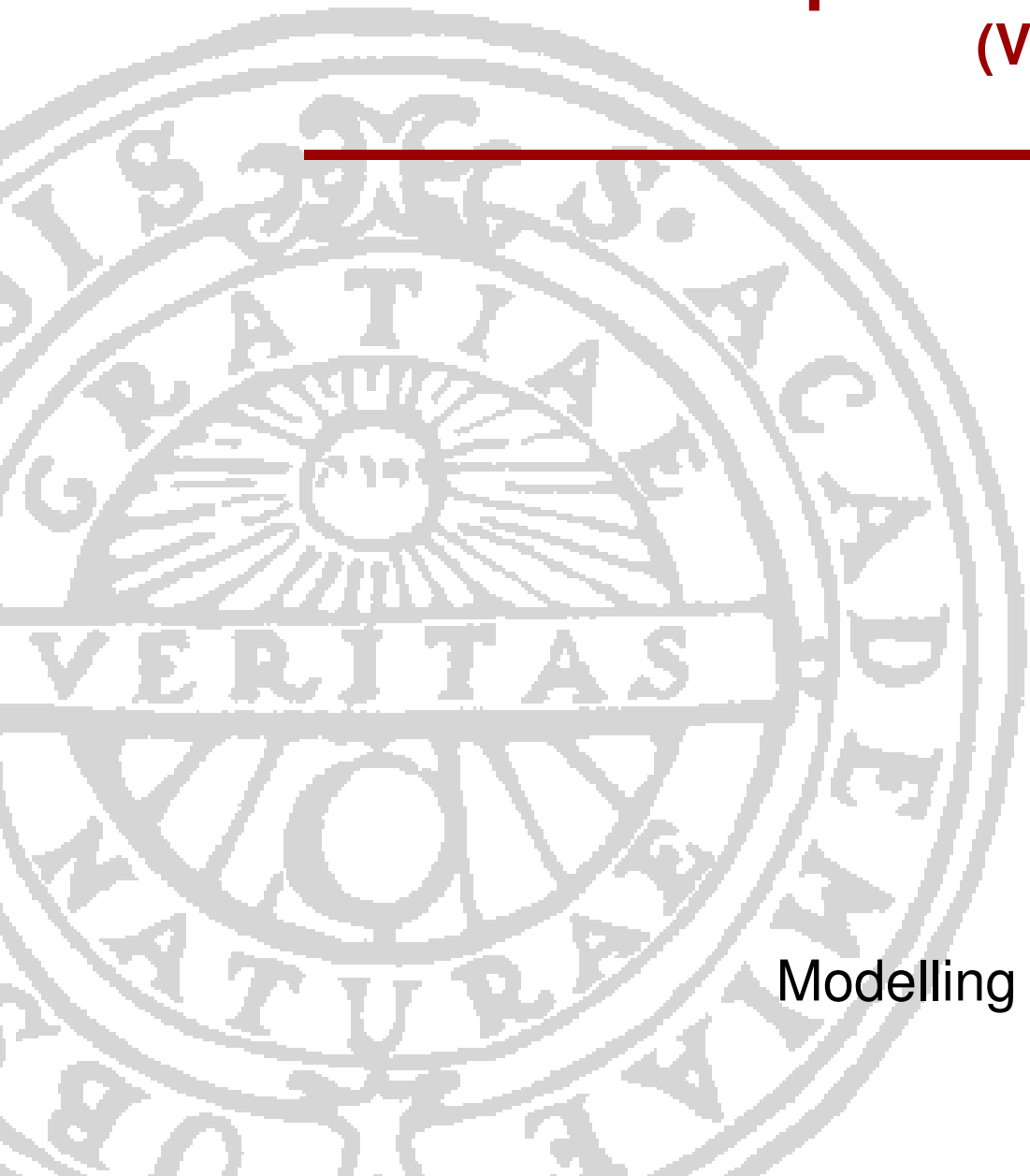


Topic 3: Constraint Predicates
(Version of 30th January 2018)

Pierre Flener, Gustav Björdal,
and Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL448:
Modelling for Combinatorial Optimisation

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Outline

Examples

Let `A` be a 1D array of variables, say with indices in `1..n`:

- The `alldifferent(A)` constraint holds if and only if (iff) all the elements of `A` take different values:
  `forall(i,j in 1..n where i < j)(A[i] != A[j]).`
- The `at_least(m,A,v)` constraint holds iff at least `m` elements of `A` take the *value* `v`, where `m` is an *integer*:
  `m <= (sum(i in 1..n)(bool2int(A[i]=v))).`
- The `count_leq(A,v,m)` constraint has the same semantics as `at_least(m,A,v)`, but `v` and `m` can even be *variables*.
- All prior uses of `count(A,v)` $\sim$ `m` had *non*-variables `v` and `m`, with $\sim \in \{<=, =, =>\}$, and should thus be reformulated respectively as `at_most(m,A,v)`, `exactly(m,A,v)`, and `at_least(m,A,v)`: always use the predicate with the most specific type signature!

UPPSALA UNIVERSITET

Definition

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

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.

UPPSALA UNIVERSITET

Motivation:

+ More compact and intuitive models, because more expressive predicates are available: islands of common combinatorial structure are identified in declarative medium-level abstractions.
+ Faster solving, due to better inference 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 variables go by many names: global-constraint predicates, combinatorial-constraint predicates, ...
- See `http://www.minizinc.org/doc-lib/doc-globals.html` and the Global Constraint Catalogue at `http://sofdem.github.io/gccat`.

UPPSALA UNIVERSITET

Outline

The `alldifferent` Predicate

Definition (Laurière, 1978)

The `alldifferent(A)` constraint, where `A` is a 1D array of variables, say with indices in `1..n`, holds iff all the elements of `A` take different values.

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

```
forall(i,j in 1..n where i < j)(A[i] != A[j])
```

Examples

- *n*-queens problem: see Topic 1: Introduction.
- Photo problem: see Topic 2: Basic Modelling.

UPPSALA UNIVERSITET

Outline

The `nvalue` Predicate

Definition (Pachet and Roy, 1999)

The `nvalue(m,A)` constraint holds iff variable `m` takes the number of distinct values taken by the elements of the 1D array `A` of variables, say with indices in `1..n`:

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

The `nvalue(A)` expression denotes the number of distinct values taken by the elements of the 1D array `A` of variables.

Note: `alldifferent(A)` iff `nvalue(n,A)`, when `A` has size `n`: always use the most specific available predicate!

Example

Model 2 of the Warehouse Location Problem:

see Topic 6: Case Studies.

UPPSALA UNIVERSITET

Outline

The `global_cardinality` Predicate

Definition (Régin, 1996)

The `global_cardinality(A,V,C)` constraint holds iff each variable `C[j]` has the number of elements of the 1D array `A` of variables that take value `V[j]`. It generalises:

- `alldifferent(A)`: set $\text{dom}(\texttt{C[j]}) = \{0, 1\}$ for all `j` and $\texttt{V} = \bigcup_{\texttt{i}=1}^{\texttt{n}} \text{dom}(\texttt{A[i]})$, if `A` has indices in `1..n`.
- `count_leq(A,v,m)`:
  set `V = [v]` and constrain `m <= C[1]`.
- `count_gt(A,v,m)`:
  set `V = [v]` and constrain `m > C[1]`.

Other variants exist: always use the most specific predicate!

Example

Model of the Magic Series problem: see Topic 4: Modelling.

UPPSALA UNIVERSITET

Outline

The `element` Predicate

Definition (Van Hentenryck and Carillon, 1988)

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

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

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

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

Use: The `element` predicate and its functional form `A[`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 int: Salary;
2 array[Jobs] of var Apps: Worker; % job j by Worker[j]
3 solve minimize sum(j in Jobs)(Salary[Worker[j]]);
4 constraint ...;          % qualifications, workload, etc
```

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

UPPSALA UNIVERSITET

Outline

The `bin_packing` Predicate

Definition

Let object O_i have the given weight or volume `V[i]`.
Let variable `B[i]` denote the bin into which O_i is put.
The `bin_packing(u,B,V)` constraint holds iff the sum of the volumes of the objects put into each bin is at most `u`; there are $\left|\bigcup_i \text{dom}(\texttt{B[i]})\right|$ bins. Variant predicates exist.

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 prerequisites under a balanced workload:

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

UPPSALA UNIVERSITET

Outline

The `knapsack` Predicate

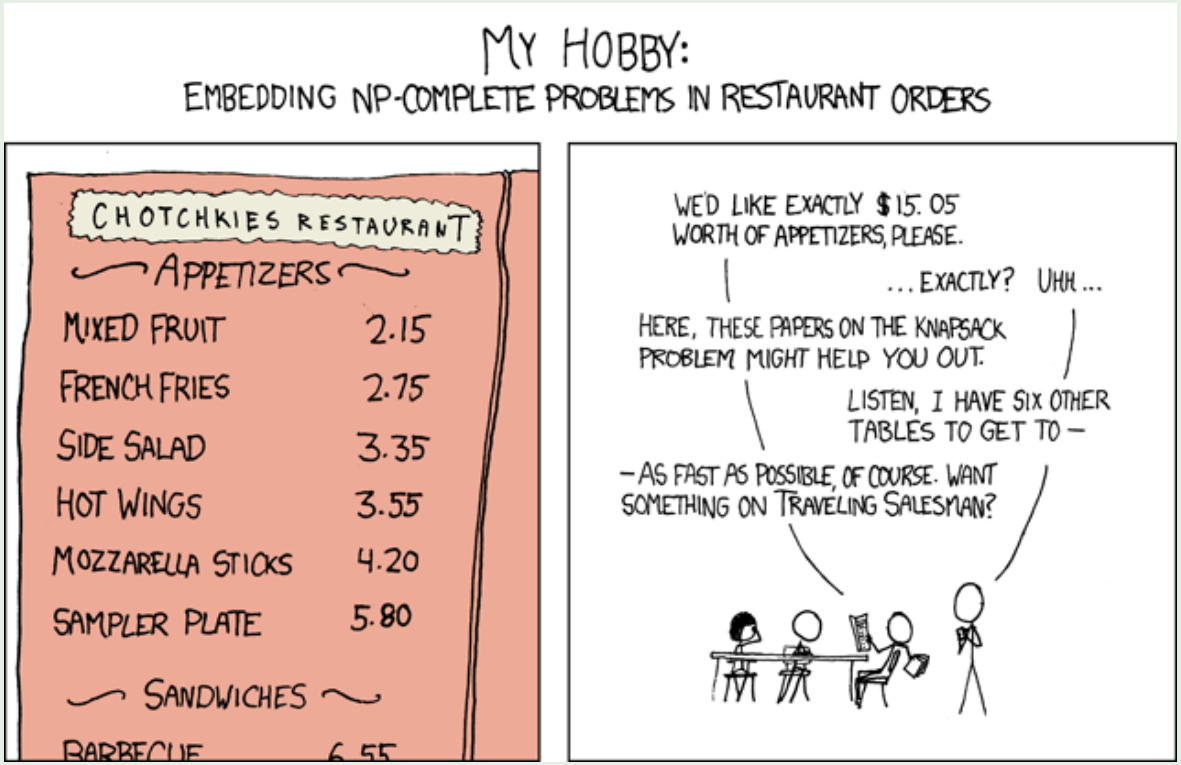
Definition

Let object type O_i have the given weight or volume `V[i]`.
Let object type O_i have the given value or profit `P[i]`.
Let the decision variable `X[i]` denote the number of copies of O_i that are put into a given knapsack.
Let the decision variables `v` and `p` respectively denote the total volume and total profit of what is in the knapsack.
Given `n` objects, the `knapsack(V,P,X,v,p)` constraint holds iff `sum(i in 1..n)(V[i] * X[i]) = v` and `sum(i in 1..n)(P[i] * X[i]) = p`.

Example

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

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



A simplified version of the Knapsack Problem, but still NP-complete.

```
1 array[1..6] of int: Cost = [215,275,335,355,420,580];
2 array[1..6] of int: Profit = [0,0,0,0,0,0];
3 array[1..6] of var 0..10: Amount;
4 constraint knapsack(Cost, Profit, Amount, 1505, 0);
5 solve satisfy;
```

See this interview for some interesting trivia.

UPPSALA UNIVERSITET

Outline

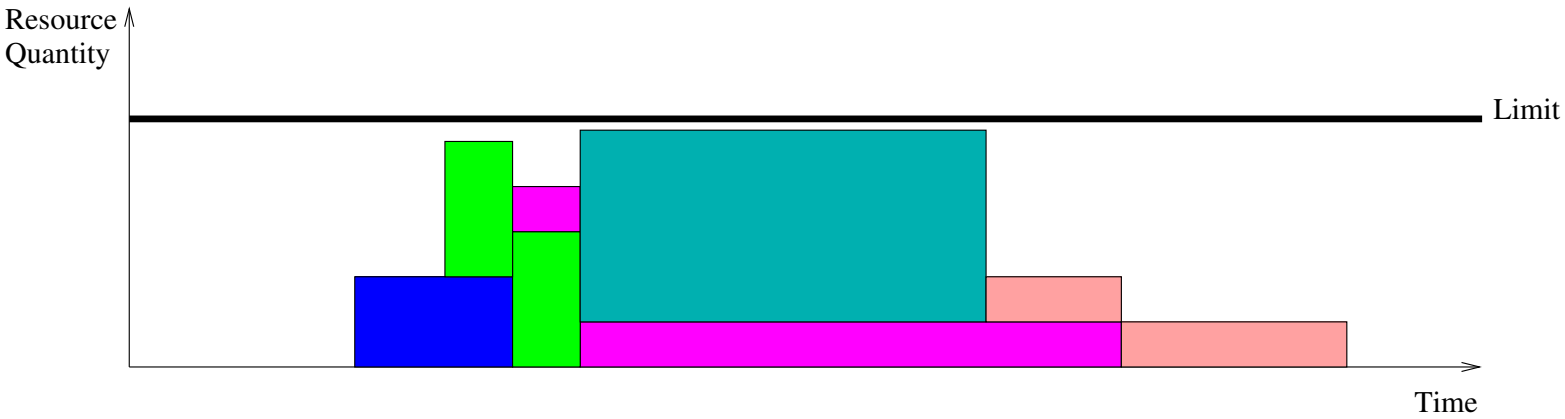
Assume we want to schedule a set of tasks to be performed over a given period such that we have the earliest end.

Definition

A task T_i is a triple $\langle S[i], D[i], R[i] \rangle$ of parameters or decision variables, where:

- `S[i]` is the starting time of task T_i
- `D[i]` is the duration of task T_i
- `R[i]` is the quantity of a global resource needed by T_i

Tasks may be run in parallel if the global resource suffices.

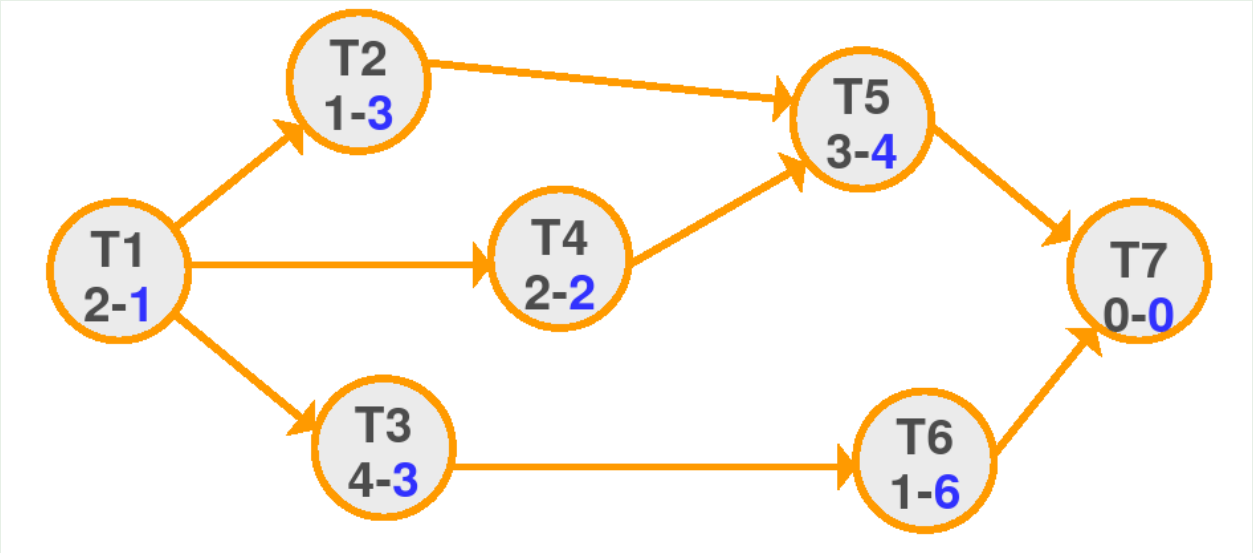


Sample schedule with parallel tasks and bounded resource

Definition

A precedence constraint of task T_1 on task T_2 expresses that the performing of T_1 must finish before T_2 can start. We say that task T_1 precedes task T_2.

Example (courtesy Magnus Ågren)



Sample tasks (bubbles), 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 finish last.

Let us temporarily ignore the bounded global resource:
If we have an unlimited global resource or each task has its own local 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 23 are modelled as follows, based on the task durations indicated there in black:

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

The `cumulative` Predicate

Definition (Aggoun and Beldiceanu, 1993)

The `cumulative(S,D,R,u)` constraint, where each task T_i has a starting time `S[i]`, a duration `D[i]`, and a resource requirement `R[i]`, ensures that the resource upper limit `u` is never exceeded when performing the T_i.

`cumulative` does not ensure any precedence constraints between the tasks: these have to be stated separately.

Example (end)

To ensure that the global resource capacity of `u = 8` units, say, is never exceeded under the resource requirements of the tasks indicated in blue on slide 23, add the following:

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

The `disjunctive` Predicate

Definition

A non-overlap constraint between tasks T_1 and T_2 states 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.

Definition (Carlier, 1982)

The `disjunctive(S,D)` constraint, where each task T_i has a starting time `S[i]` and a duration `D[i]`, ensures that no tasks T_i and T_j overlap. It has 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

The `circuit` and `subcircuit` Predicates

Enabling the representation of a circuit in a digraph:

- Let variable `S[i]` represent the successor of vertex `i`.
- The domain of `S[i]` is the set of indices `j` such that there is an arc from vertex `i` to vertex `j`, plus `i` itself.

Definition (Laurière, 1978)

The `circuit(S)` constraint holds iff the arcs $\texttt{i} \rightarrow \texttt{S[i]}$ form a Hamiltonian circuit: each vertex is visited once.

Definition (Beldiceanu and Contejean, 1994)

The `subcircuit(S)` constraint holds iff `circuit(S')` holds for exactly one possibly empty non-singleton subarray `S'` of `S`, and $\texttt{S[i]} = \texttt{i}$ for all the other vertices.

Examples

Assume the successor variables in `S` take these values:

- `[2,3,4,1]`: circuit $1 \to 2 \to 3 \to 4 \to 1$.
- `[3,1,2,4]`: subcircuit $1 \to 3 \to 2 \to 1$, and `S[4]=4`.
- `[1,2,3,4]`: empty subcircuit, and `S[i]=i` for all `i`.
- `[3,4,1,2]`: not a solution, as there are two subcircuits, namely $1 \to 3 \to 1$ and $2 \to 4 \to 2$.

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

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

Requiring a path from vertex `i` to vertex `j`:

```
constraint subcircuit(S) /\ S[j] = i;
```

upon adding `i` to the domain of `S[j]` if need be.

UPPSALA UNIVERSITET

Outline

The `lex_lesseq` Predicate

Example

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

Definition

The `lex_lesseq(A,B)` constraint, where `A` and `B` are same-length 1D arrays of variables, say both with indices in `1..n`, holds iff `A` is lexicographically at most equal to `B`:

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

Variant predicates exist.

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

UPPSALA UNIVERSITET

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.

Regular Expressions and Languages

Definition

Let Σ be an alphabet, that is a finite set of symbols.
Regular expressions over Σ are defined as follows:

- $\varnothing$ is a regular expression: its language, $\mathcal{L}(\varnothing)$, is $\varnothing$.
- 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: $\mathcal{L}(r^*) = \{w^i \mid i \in \mathbb{N} \wedge w \in \mathcal{L}(r)\}$.

A regular expression defines a regular language over Σ.

Regular Expressions

Common abbreviations for regular expressions:

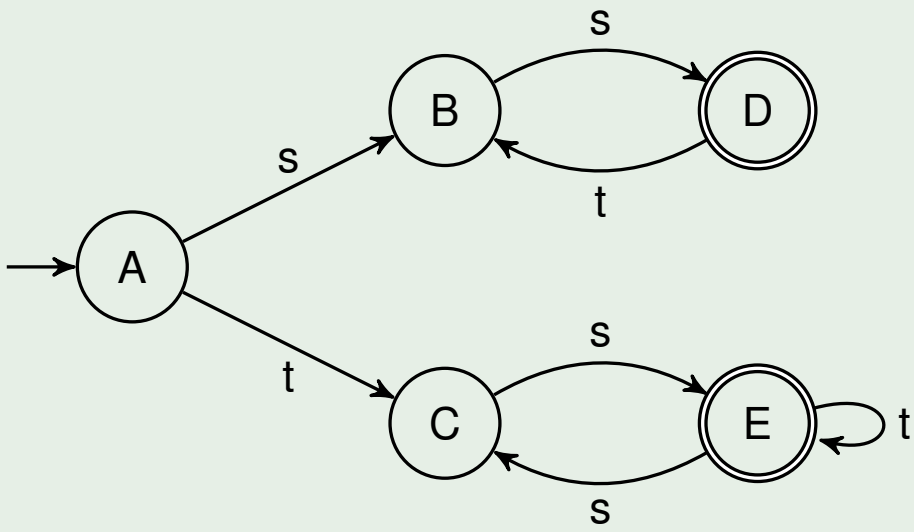
Let r be a regular expression:

- [**abcd**] denotes **a**|**b**|**c**|**d**
- [**b-g**] denotes [**bcdefg**]
- [**a-cA-C01**] denotes [**abcABC01**]
- $r^?$ denotes $r|\epsilon$
- r^+ denotes rr^*
- r^4 denotes $rrrr$

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)

Example



Conventions:

- Start state: A (marked by arc coming in from nowhere).
- Accepting states: D and E (marked by double circles).
- Determinism: There is one outgoing arc per symbol in alphabet $\Sigma = \{s, t\}$; missing arcs go to a non-accepting state that has self-loops on every symbol in Σ.

The `regular` and `table` Predicates

Definition (Pesant, 2004)

The `regular(A,D)` constraint holds iff the values taken by the 1D variable array `A` form a string that belongs to the regular language accepted by the DFA `D`.

Use the transformation algorithms of automata theory to convert a regular expression into a (minimised) DFA.

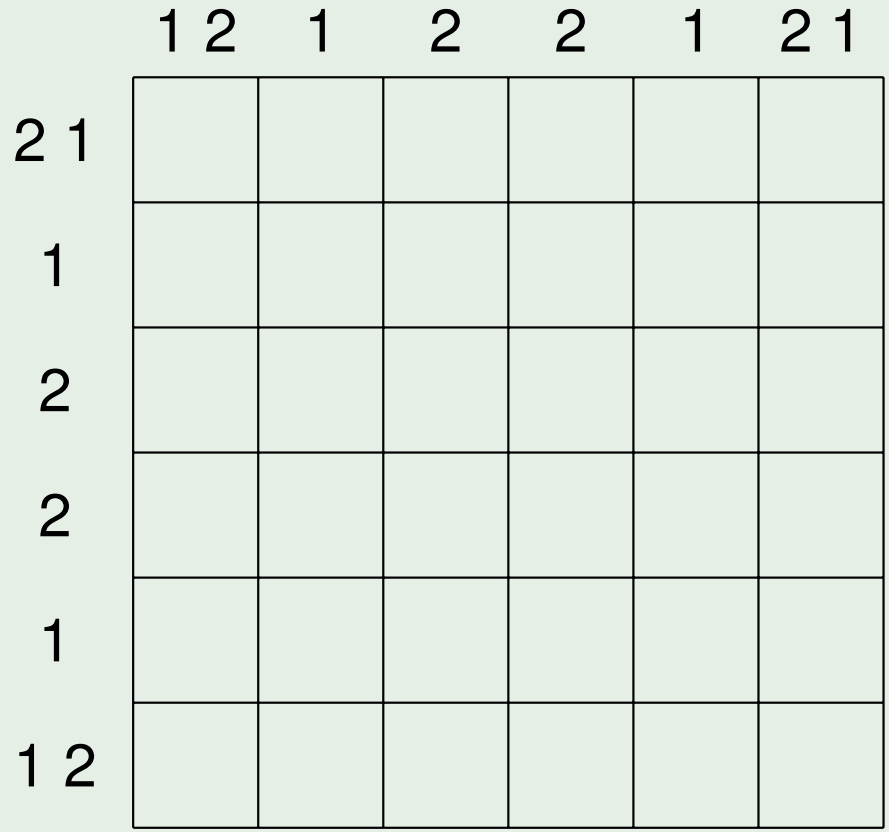
Definition

The `table(A,T)` constraint holds iff the values taken by the 1D variable array `A` form a row of the 2D value array `T`.

The 2D array `T` gives an extensional definition of a new constraint predicate, as opposed to the intensional definition given so far for all other constraint predicates.

Example (The Nonogram Puzzle: instance)

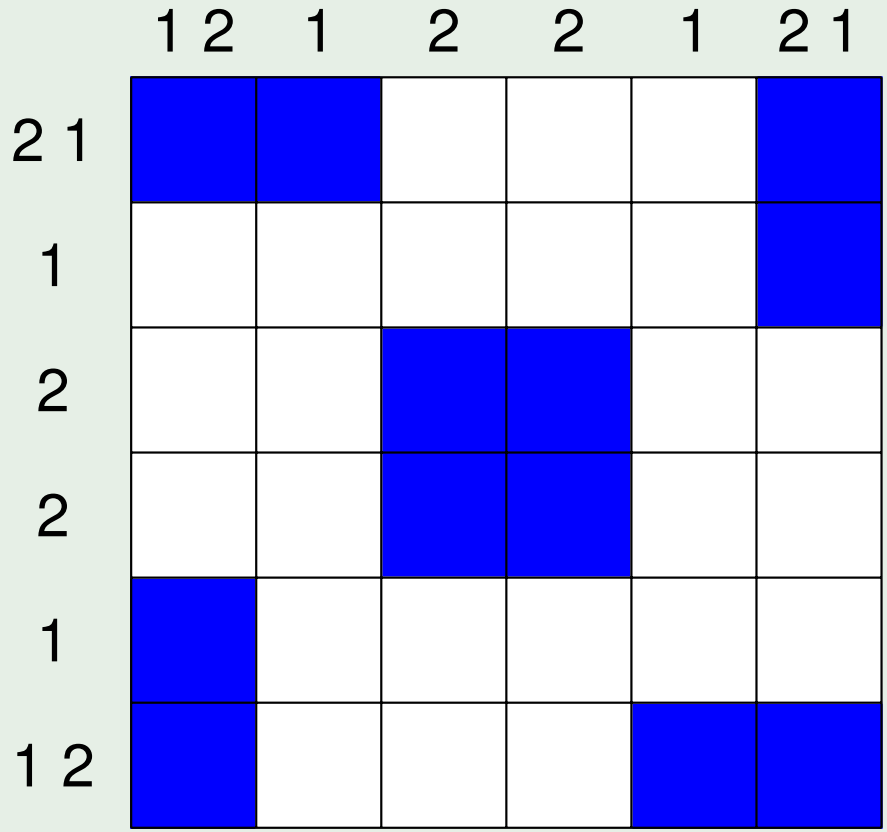
Each hint gives the sequence of lengths of blue blocks in its row or column, with at least one white cell between blocks, but possibly none before the first and after the last block.



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 and after the last block.

Solution:



Example (The Nonogram Puzzle: model)

Model:

- Decision variables: An enumeration-type variable for each cell, with value `w` if it is to be coloured white, and value `b` if it is to be coloured blue.
- Constraints: State a `regular` constraint for each hint. For example, for hint 2 3 1 on a row or column `A` of length $n \geq 8$, state `regular(A,` $w^* b^2 w^+ b^3 w^+ b^1 w^*$`)`, but replace the regular expression by a (minimised) DFA for the same language.

See Survey of Paint-by-Number Puzzle Solvers: the straightforward model above fares well, at least with a CP solver, compared to hand-written problem-specific code.

Example (Nurse Rostering)

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

- `n` normal shift (this value is not available on Sundays)
- `l` long shift (this value is not available on Sundays)
- `s` Sunday shift (this value is only available on Sundays)
- `o` day off

The nurse labour union imposes the following regulations:

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

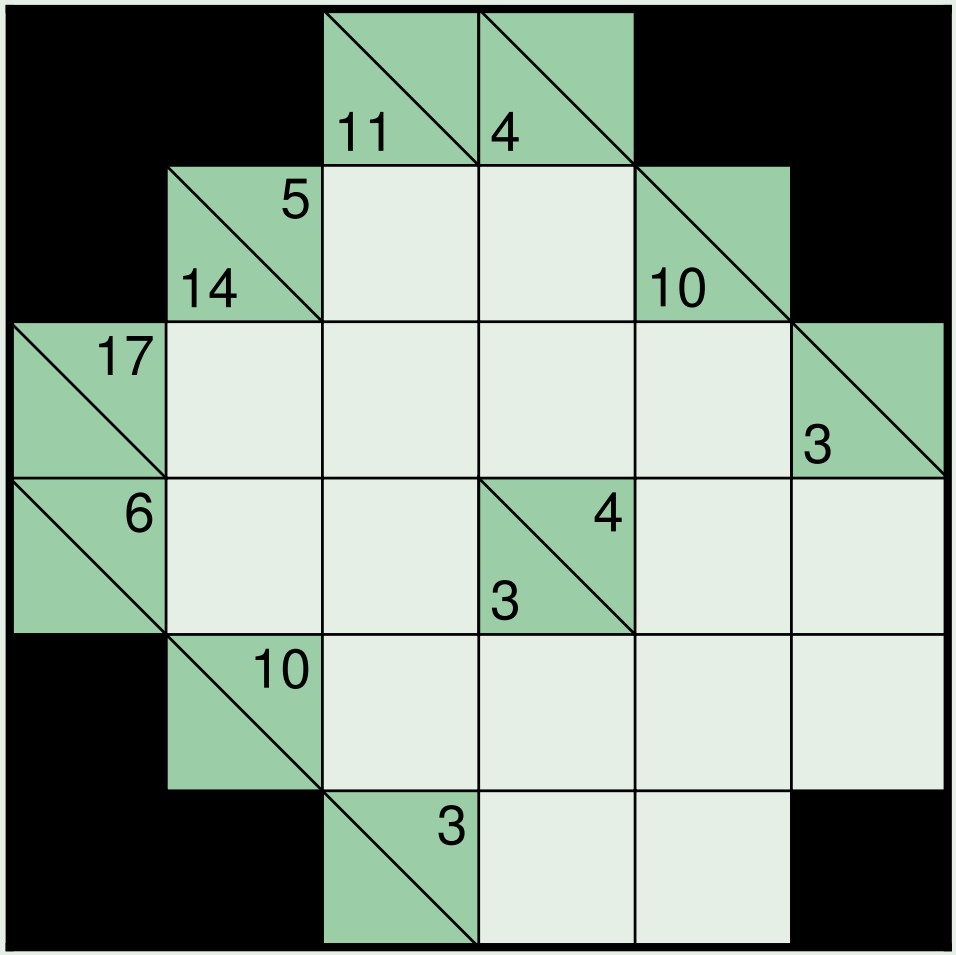
For each nurse, state the following constraint over the scheduling horizon, say 17 weeks here:

```
regular([Sun1,Mon1,...,Fri17,Sat17],(so|llo|n|o)*)
```

Further, a hospital has constraints on nurse presence.

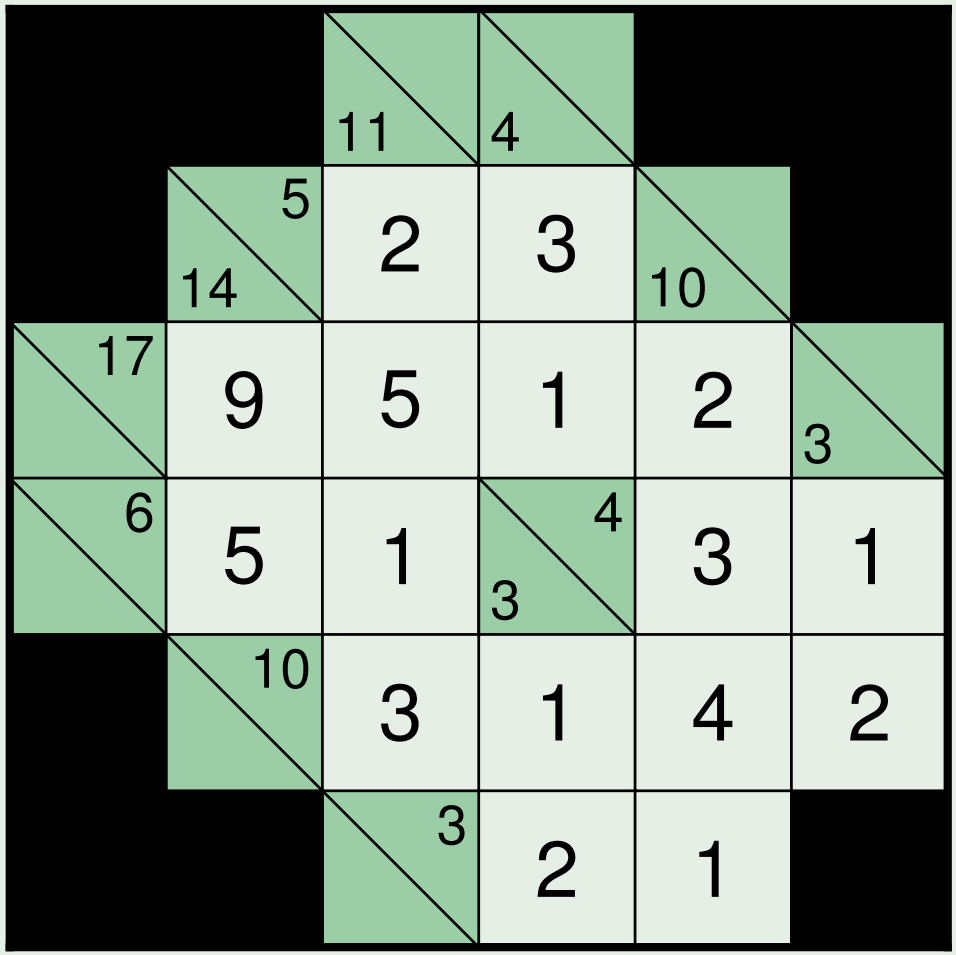
Example (The Kakuro Puzzle: instance)

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



Example (The Kakuro Puzzle: first model)

Model:

- Decision variables: An integer variable for each cell, with domain `1..9`.
- Constraints: For each hint `A[`α`] + ··· + A[`β`] =` σ, state `alldifferent(i in `α`..`β`)(A[i]) /\ sum(i in `α`..`β`)(A[i]) =` σ.

Performance, using a CP solver:

- 22 $\times$ 14 Kakuro with 114 hints: 9638 nodes, 160 s
- 90 $\times$ 124 Kakuro with 4558 hints: ? nodes, ? years

Symptom: The decomposition may give weak inference: for `x!=y /\ x+y=4`, CP inference gives `x,y in 1..3`, not noticing that `2` should be pruned from both domains. We may need a custom predicate `alldifferent_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 `alldifferent` and `sum`-based constraints of each hint?

- For the hint `x+y=4`, state `regular([x,y],`**13**|**31**`)`.
- For the hint `y+z=3`, state `regular([y,z],`**12**|**21**`)`.
- In MiniZinc, one cannot provide a regular expression, but one must convert it into a (minimised) DFA. If, as above, only concatenation (rs) and alternation ($r|s$) are used, then one can also use `table` instead:

```
table([x,y],[|1,3|3,1|]) /\
table([y,z],[|1,2|2,1|]).
```

- What about the hint $\texttt{A[}\alpha\texttt{]} + \cdots + \texttt{A[}\alpha\texttt{+8]} = 45$? There are $9! = 362,880$ solutions...

Example (The Kakuro Puzzle: second model, end)

New model (end):

- For the hint `A[`α`] + ··· + A[`α`+8] = 45`, it suffices to state `alldifferent(i in `α`..`α`+8)(A[i])`, as the sum of 9 distinct non-0 digits is necessarily 45.
- For the hint `A[`α`]+···+A[`α`+7] =` σ, it suffices to state `alldifferent([A[i]|i in `α`..`α`+7]++[45-`σ`])`.
- For the hint `A[`α`] =` σ, it suffices to state `A[`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 4558 hints: 0 nodes, 345 ms!

The Kakuro story is based on material by Christian Schulte.

UPPSALA UNIVERSITET

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 variables is not a built-in of the used solver.
- A definition of γ in terms of built-in predicates is not obvious to the modeller, or it has turned out that its inference is too expensive or weak.
- The modeller does not have the time or skill to design an inference algorithm for γ, or deems γ not reusable.
- The complexity and strength of an inference algorithm for γ are not deemed crucial for the time being.

Important Modelling Device

Example (Encoding a small function)

The constraint `x * x = y`, where there is exactly one `y` for every `x`, may yield poor inference: for `x` in `1..6`, say, try `element(x,[1,4,9,16,25,36],y)`, that is `[1,4,9,16,25,36][x] = y`, for better inference.

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

Example (Encoding a small relation)

The constraint `x * x = abs(y)`, where there can be more than one `y` for every `x`, and vice-versa, may yield poor inference: for `x` 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 inference (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.
Intro. to Automata Theory, Languages, & Computation.
Third edition. Addison-Wesley, 2007.