U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

Logic and Constraint Programming

Masters in Informatics and Computing Engineering
2022/2023 – 2nd Semester

Constraint Logic Programming using SICStus Prolog

SICStus Prolog User's Manual (Release 4.8)

Section 10.10: Constraint Logic Programming over Finite Domains—library(clpfd)

Partially based on previous slides from Henrique Lopes Cardoso (hlc@fe.up.pt), Luís Paulo Reis (lpreis@fe.up.pt), other authors and the SICStus Prolog manual (v. 4.8)

26/03/2023

Daniel Castro Silva
dcs@fe.up.pt

Content

CLP in SICStus Prolog

1. AVAILABLE CONSTRAINT DOMAINS

Boolean and Real Domains

- Booleans:
  - clp(B) Scheme
  - `use_module(library(clpb)).`
    - Section 10.9 of SICStus manual

- Reals and Rationals
  - clp(Q,R) Scheme
  - `use_module(library(clpq)).` `use_module(library(clpr)).`
    - Section 10.11 of SICStus manual

- Not seen in detail in this curricular unit

Finite Domains

- ***clp(FD)*** solver is an instance of the general CLP schema introduced in [Jafar & Michaylov 87].
- Useful to model discrete optimization problems
  - Scheduling, planning, resource allocation, packing, timetabling, ...
- ***clp(FD)*** solver characteristics:
  - Two constraint classes: primitive and global
  - Very efficient propagators for global constraints
  - The logical value of a primitive constraint can be reflected into a binary variable (0/1) – materialization (or reification)
  - New primitive constraints can be added, by writing indexicals
  - New global constraints can be written in Prolog

CLP in SICStus Prolog

2. CLP(FD) SOLVER INTERFACE

CLP(FD) Solver Interface

- The ***clp(FD)*** solver is available as a library

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

- It contains predicates to test the consistency and entailment of constraints over finite domains, as well as to obtain solutions by attributing values to the variables

- A **finite domain** is a subset of small integers and a **constraint over finite domains** is a relation between a tuple of small integers
- Only small integers and non-instantiated variables are allowed in constraints over finite domains
  - Small integer: $[-2^{28}, 2^{28}-1]$ in 32-bit platforms, or $[-2^{60}, 2^{60}-1]$ in 64-bit platforms
    - You can use the *prolog_flag/2* predicate to obtain these values

CLP(FD) Solver Interface

- All **domain variables** have an associated finite domain, explicitly declared in the program or implicitly imposed by the solver
  - Temporarily, a variable's domain can be infinite, if it doesn't have a finite lower or upper bound
  - The domain of a variable reduces as constraints are added
- If a domain becomes empty, the constraints are not satisfiable as a whole, and the current computing branch fails
- At the end of the computation, it is usual for each variable to have its domain constrained to a single value (singleton)
  - Typically, some search is required for this to happen
- Each constraint is implemented by a (set of) propagator(s)
  - Indexicals
  - Global propagators

Structure of a CLP Program

- A CLP program is structured in the following three steps:
  - Declaration of variables and respective domains
  - Declaration of constraints over the variables
  - Search for a solution

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

example:-
        A in 1..7,
        domain([B, C], 1, 10),
        A + B + C #= A * B * C,
        A #> B,
        labeling([], [A, B, C]).
```

variables and domains

constraints

search for solution

```
| ?- example.
A = 2,
B = 1,
C = 3 ?
```

Structure of a CLP Program

- The order of these steps is important
  - If we invert the order, by placing the search for a solution first and only then the constraints, we end up with the much less efficient traditional Generate & Test mechanism

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

badExample:-
       A in 1..7,
       domain([B, C], 1, 10),
       labeling([], [A, B, C]),
       write('.'),
       A + B + C #= A * B * C,
       A #> B.
```

```
| ?- badExample.
..........................
..........................
..........................
.........................
A = 2,
B = 1,
C = 3 ?
```

Variable Domains

- A variable may have its domain declared using ***in/2*** and a range of values, given by a *ConstantRange*:
  - `GradePLR in 16..20`
  - The definition of *ConstantRange* allows for the declaration of much more complex domains
  - *`VarA in (2..8) \/ (15..20)`*
  - *`VarB in {4, 8, 15, 16, 23, 42}`*

ConstantSet	::= `{integer,…,integer}`
ConstantRange	::= *ConstantSet*
	| *Constant* .. *Constant*
	| *ConstantRange* /\ *ConstantRange*
	| *ConstantRange* \/ *ConstantRange*
	| \ *ConstantRange*

Variable Domains

- You may also use ***in_set/2*** to declare the domain of a variable
  - The second argument of ***in_set/2*** is a *Finite Domain Set*, which can be obtained from a list by using the ***list_to_fdset(+List, -FD_Set)*** predicate
    - See section 10.10.9.3 for operations over *FD Sets*

```
Numbers = [4, 8, 15, 16, 23, 42],
list_to_fdset(Numbers, FDS_Numbers),
Var in_set FDS_Numbers.
```

Variable Domains

- The ***domain(+List_of_Variables, +Min, +Max)*** predicate can be used to declare a simple domain for a list of variables:
  - `domain([A, B, C], 5, 12)`

- Other constraints limit the domains of the variables involved
  - *`A #> 8`*
  - *`B + C #< 12`*
  - *`A + B + C #= 20`*

Posting Constraints

- A constraint is called just like any other Prolog predicate

```
| ?- X in 1..5,
       Y in 1..3,
       X * Y #= 20.
no
```

```
| ?- X in 1..5,
       Y in 2..8,
       X+Y #= T.
X in 1..5,
Y in 2..8,
T in 3..13
```

```
| ?- X in 1..5,
       T in 3..13,
       X+Y #= T.
X in 1..5,
T in 3..13,
Y in -2..12
```

- The existence of an answer is an indication that, after filtering and propagation, all variables have valid domains (at least 1 possible value)
  - This, by itself, is no guarantee that a solution to the problem exists
- The constraints associated to each variable are not shown

Posting Constraints

- By posting a constraint, the propagation mechanism is called, which limits variable domains
  - This mechanism can be computationally heavy in some cases
- It is possible to post a set of constraints at once (in batch), suspending the propagation mechanism until all these constraints have been placed, using the ***fd_batch(+Constraints)*** predicate
  - *Constraints* is a list with the constraints to be placed

```
| ?- domain([A,B,C], 5, 12),
     A #> 8,
     B+C #< 12,
     A+B+C #= 20.
A in 9..10,
B in 5..6,
C in 5..6 ?
```

```
| ?- domain([A,B,C], 5, 12),
     fd_batch( [A #> 8,
         B+C #< 12,
         A+B+C #= 20] ).
A in 9..10,
B in 5..6,
C in 5..6 ?
```

Forgetting Constraints

- Variables and associated constraints can also be 'forgotten' using the ***fd_purge(+Variable)*** predicate
  - Note that this erases the variable and all associated constraints

```
| ?- domain([A,B,C], 1, 10),
     all_distinct([A, B, C]),
     A+B+C #= A*B*C,
     labeling([], [A,B,C]).
A = 1,
B = 2,
C = 3 ?
```

```
| ?- domain([A,B,C], 1, 10),
     all_distinct([A, B, C]),
     A+B+C #= A*B*C,
     fd_purge(C),
     labeling([], [A,B]).
A = 1,
B = 1 ?
```

Materialized (Reified) Constraints

- Sometimes it is useful to reflect the truth value of a constraint into a boolean variable B (0/1) such that:
  - The constraint is placed if B has value 1
  - The negation of the constraint is placed if B has value 0
  - B is set to 1 if the constraint is *entailed*
  - B is set to 0 if the constraint is *disentailed*
- This mechanism is known as **materialization** or **reification**
- A materialized constraint is written in the form

  Constraint #<=> B.

  where *Constraint* is a reifiable constraint and B a binary variable

Materialized (Reified) Constraints

- Example: ***exactly(X, L, N)***
  - True if ***X*** occurs exactly ***N*** times in list ***L***
  - It can be recursively defined

```
exactly(_, [], 0).
exactly(X, [Y|L], N):-
     X #= Y #<=> B,
     N #= M + B,
     exactly(X, L, M).
```

- Reifiable constraints can be used as terms in arithmetic expressions:

```
| ?- X #= 10,
     B #= (X#>=2) + (X#>=4) + (X#>=8).
B = 3,
X = 10
```

```
| ?- X in 1..3,
     B #= (X#>=1) + (X#>=2) + (X#>=3),
     labeling([], [X]).
X = 1, B = 1 ? ;
X = 2, B = 2 ? ;
X = 3, B = 3 ? ;
no
```

CLP in SICStus Prolog

3. AVAILABLE CONSTRAINTS

Available Constraints

- Arithmetic Constraints
- Membership Constraints
- Propositional Constraints
- Combinatorial Constraints
  - Arithmetic-Logical
  - Scheduling
  - Placement
  - Graph
  - Sequence
  - Extensional

Arithmetic Constraints

- ***Expr RelOp Expr***
  - ***RelOp***: *#= | #\= | #< | #=< | #> | #>=*
  - The expressions can be linear or non-linear
  - Linear expressions lead to a better propagation
    - For instance, ***X/Y*** and ***X mod Y*** block until ***Y*** is ground
  - Linear arithmetic constraints maintain interval consistency
  - Arithmetic constraints can be materialized
  - Example:

```
| ?- X in 1..2,
     Y in 3..5,
     X#=<Y #<=> B.
B = 1,
X in 1..2,
Y in 3..5
```

Sum

- ***sum(Xs, RelOp, Value)***
  - ***Xs*** is a list of integers and/or domain variables, ***RelOp*** is a relational operator and ***Value*** is an integer or domain variable
  - The constraint holds if *sum(Xs) RelOp Value* (the sum of all elements in ***Xs*** has relation ***RelOp*** to ***Value***)
  - It approximates *sumlist/2* from the *lists* library (when *RelOp* is #=)
  - Cannot be materialized
  - Examples:

```
| ?- domain([X,Y], 1, 10),
     sum([X,Y], #<, 10).
X in 1..8,
Y in 1..8
```

```
| ?- domain([X,Y], 1, 10),
     sum([X,Y], #=, Z).
X in 1..10,
Y in 1..10,
Z in 2..20
```

Scalar Product

- ***scalar_product(Coeffs, Xs, RelOp, Value)***
- ***scalar_product(Coeffs, Xs, RelOp, Value, Options)***
  - ***Coeffs*** is a list of length *n* of integers, ***Xs*** a list of length *n* of integers and/or domain variables, ***RelOp*** a relational operator and ***Value*** an integer or domain variable
  - The constraint holds if *sum(Coeffs*Xs) RelOp Value*
  - This constraint is materializable
  - ***Options*** is a list of options
    - ***among(Least, Most, Range)*** imposes that at least ***Least*** and at most ***Most*** elements from ***Xs*** must have values within ***Range*** (a *ConstantRange*)
    - ***consistency(Cons)*** denotes the level of consistency to be used by the constraint. ***Cons*** can take the values ***domain***, ***bounds*** or ***value*** (by default, bounds consistency is maintained).

Scalar Product

- ***scalar_product_reif(Coeffs, Xs, RelOp, Value, Reif)***
- ***scalar_product_reif(Coeffs, Xs, RelOp, Value, Reif, Options)***
  - Reified version of *scalar_product/[4,5]*.
  - Equivalent to materializing the previous constraint
  - Examples:

```
| ?- domain([A,B,C], 1, 5),
     scalar_product([1,2,3], [A,B,C], #=, 10).
A in 1..5,
B in 1..3,
C in 1..2

| ?- domain([A, B, C], 1, 5),
     scalar_product_reif([1,2,3], [A,B,C], #<, 6, R).
R = 0,
A in 1..5, B in 1..5, C in 1..5 ?
```

Minimum / Maximum

- ***minimum(Value, Xs)***
- ***maximum(Value, Xs)***
  - ***Xs*** is a list of integers and/or domain variables
  - ***Value*** is an integer or domain variable
  - The constraint holds if ***Value*** is the minimum (or maximum) value of ***Xs***
  - Corresponds to *min_member/2* (*max_member/2*) from *library(lists)*
  - Cannot be materialized
  - Examples:

```
| ?- domain([A,B], 1, 10),
     C in 5..15,
     minimum(C, [A,B]).
A in 5..10,
B in 5..10,
C in 5..10
```

```
| ?- domain([A,B,C], 1, 5),
     sum([A,B,C], #=, 10),
     maximum(3, [A,B]).
A in 2..3,
B in 2..3,
C in 4..5
```

Minimum_arg / Maximum_arg

- ***minimum_arg(Xs, Index)***
- ***maximum_arg(Xs, Index)***
  - ***Xs*** is a list of integers and/or domain variables
  - ***Index*** is an integer or domain variable
  - The constraint holds if ***Index*** is the index of the minimum (maximum) value of ***Xs***
  - If the value appears more than once in ***Xs***, ***Index*** points to the first occurrence
  - Cannot be materialized
  - Examples:

```
| ?- minimum_arg([3,1,2,5], A).
A = 2

| ?- maximum_arg([3,1,2,5], B).
B = 4
```

```
| ?- domain([A,B,C], 1, 5),
     sum([A,B], #=, 10),
     minimum_arg([A,B,C], X).
A = 5,
B = 5,
C in 1..5
X in {1} \/ {3}
```

If Then Else

- ***if_then_else(If, Then, Else, Value)***
  - ***If*** is an integer or domain variable (can only take values 0 or 1)
  - ***Then***, ***Else***, and ***Value*** are integers or domain variables
  - The constraint holds when ***If*** = 1 and ***Value*** = ***Then***, or ***If*** = 0 and ***Value*** = ***Else***
  - Corresponds to a typical if-then-else construct
  - Example:

```
| ?- domain([A,B,C], 1, 2),
     A #\= B,
     A #> B #<=> D,
     if_then_else(D, A, B, C),
     labeling([], [A,B,C]).
A = 1,
B = 2,
C = 2,
D = 0 ? ;
```

```
A = 2,
B = 1,
C = 2,
D = 1 ? ;
no
```

Membership Constraints

- Predicates used to define variable domains
  - ***domain(+Vars, +Min, +Max)***
    - True if all elements in ***Vars*** are in the interval ***Min..Max***
  - ***?X in +Range***
    - True if ***X*** is an element of the *ConstantRange* ***Range***
  - ***?X in_set +FDSet***
    - True if ***X*** is an element of the set ***FDSet***
  - *in/2* and *in_set/2* maintain domain consistency and are materializable
  - Examples:

```
| ?- domain([X], 1, 3),
     X in 3..5 #<=> B,
     labeling([], [X]).
X = 1, B = 0 ? ;
X = 2, B = 0 ? ;
X = 3, B = 1 ? ;
no
```

```
| ?- X in {1, 2, 3, 5}.
X in (1..3)\/{5}

| ?- list_to_fdset([1,2,3,5], FD),
     X in_set FD.
FD = [[1|3], [5|5]],
X in (1..3)\/{5}
```

Propositional Constraints

- Allow for the definition of propositional formulas over materializable constraints
- Example: `X #= 2 #\/ Y #= 4`
  - Expresses the disjunction between two equality constraints
- The leafs of the propositional formulas can be materializable constraints, constants 0 and 1, or boolean variables (domain 0/1)
- These constraints maintain domain consistency (even though domains are not affected by posting propositional constraints)
- New primitive materializable constraints can be defined using indexicals
- Example:

```
| ?- X in 1..2, Y in 1..10,
     X #= Y #\/ Y #< X,
     labeling([], [X,Y]).
X = 1, Y = 1 ? ;
X = 2, Y = 1 ? ;
X = 2, Y = 2 ? ;
no
```

```
| ?- X in 1..2, Y in 1..10,
     X #= Y #\/ Y #< X.
X in 1..2,
Y in 1..10 ?

| ?- A in 1..10, A #= 2 #\/ A #= 4.
A in 1..10 ?
```

Propositional Constraints

#\ :Q	true if constraint Q is false (NOT)
:P #/\ :Q	true if constraints P and Q are both true (AND)
:P #\ :Q	true if exactly one of the constraints P and Q is true (XOR)
:P #\/ :Q	true if at least one of the constraints P and Q is true (OR)
:P #=> :Q :Q #<= :P	true if constraint Q is true or if constraint P is false (implication)
:P #<=> :Q	true if P and Q are both true or both false (equivalence)

- Note that materialization is a particular case of the equivalence propositional constraint (when Q is a variable with domain 0/1)

Combinatorial Constraints

- Combinatorial Constraints are also called symbolic constraints
- Usually they maintain interval consistency on the variables involved

Arithmetic-Logical

- *smt/1 (deprecated, see case)*
- *count/4 (deprecated)*
- *global_cardinality/[2,3]*
- *nvalue/2*
- *all_equal/1, all_equal_reif/2*
- *all_different/[1,2]*
- *all_distinct/[1,2]*
- *all_different_except_0/1*
- *all_distinct_except_0/1*
- *symmetric_all_different/1*
- *symmetric_all_distinct/1*
- *assignment/[2,3]*
- *sorting/3*
- *keysorting/[2,3]*
- *lex_chain/[1,2]*
- *bool_[and,or,xor]/2*
- *bool_channel/4*

Scheduling

- *cumulative/[1,2]*
- *cumulatives/[2,3]*
- *multi_cumulative/[2,3]*

Placement

- *bin_packing/2*
- *disjoint1/[1,2]*
- *disjoint2/[1,2]*
- *diffn/[1,2]*
- *geost/[2,3,4]*

Graph

- *circuit/[1,2]*
- *subcircuit/[1,2]*

Sequence

- *automaton/[3,8,9]*
- *regular/2*
- *value_precede_chain/[2,3]*
- *seq_precede_chain/[1,2]*

Extensional

- *element/[2,3]*
- *relation/3 (deprecated)*
- *table/[2,3]*
- *case/[3,4]*

Count

(*deprecated*, see *global_cardinality*)

- ***count(+Val, +List, +RelOp, ?Count)***
  - Constrains the number of occurrences of a given value within a list
  - ***Val*** is an integer, ***List*** a list of integers and/or domain variables, ***Count*** an integer or domain variable, and ***RelOp*** a relational operator
  - The constraint holds if the number of occurrences of ***Val*** within ***List*** has relation ***RelOp*** with ***Count***
  - Maintains domain consistency, but ***global_cardinality/2*** is a better alternative
  - Examples:

```
| ?- domain([X,Y,Z], 1, 3),
     count(1, [X,Y,Z], #>, Z).
X in 1..3,
Y in 1..3,
Z in 1..2
```

```
| ?- domain([A,B,C], 1, 3), X in 2..5,
     count(1, [A,B,C], #=, X),
     labeling([], [X]).
X = 2, A in 1..3, B in 1..3, C in 1..3 ? ;
A = 1, B = 1, C = 1, X = 3 ? ;
no
```

Global Cardinality

- ***global_cardinality(+Xs, +Vals)***
- ***global_cardinality(+Xs, +Vals, +Options)***
  - Constrains the number of occurrences of each value within a list of variables
  - ***Xs*** is a list of integers and/or domain variables; ***Vals*** is a list of ***K-V*** terms, where ***K*** is a unique integer and ***V*** is an integer or a domain variable
  - True if each element in ***Xs*** equals one of ***K*** and for each pair ***K-V*** exactly ***V*** elements of ***Xs*** equal ***K***
  - If ***Xs*** or ***Vals*** are ground, and in other special cases, it maintains domain consistency; interval consistency may not be assured
  - ***Options*** is a list of options (to control propagation) (see documentation)
  - Examples:

```
| ?- global_cardinality([A,B,C],
              [1-2, 3-1]).
A in {1}\/{3},
B in {1}\/{3},
C in {1}\/{3}
```

```
| ?- A in 3..10,
     global_cardinality([A,B,C],
              [1-2, 3-1]).
A = 3, B = 1, C = 1
```

Nvalue

- ***nvalue(?N, +Variables)***
  - Constrains the list of variables ***Variables*** in such a way that there are exactly ***N*** distinct values
  - ***Variables*** is a list of integers and/or domain variables with finite limits and ***N*** is an integer or domain variable
  - Can be seen as a relaxed version of ***all_distinct/2***
  - Examples:

```
| ?- domain([X,Y], 1, 3),
     domain([Z], 3, 5),
     nvalue(2, [X,Y,Z]),
     X #\= Y, X #= 1.
X = 1,
Y = 3,
Z = 3
```

```
| ?- domain([X,Y], 1, 3),
     domain([Z], 1, 5),
     nvalue(2, [X,Y,Z]),
     X#\=Y, X#=1.
X = 1,
Y in 2..3,
Z in 1..3
```

```
| ?- domain([X,Y], 1, 3),
     domain([Z], 1, 5),
     nvalue(2, [X,Y,Z]),
     X#\=Y.
X in 1..3,
Y in 1..3,
Z in 1..5
```

All Equal

- ***all_equal(+Variables)***
- ***all_equal_reif(+Variables, B)***
  - ***Variables*** is list of integers and/or domain variables, ***B*** a boolean variable/integer
  - The constrain holds if all values in the ***Variables*** list are the same
    - Equivalent to a ***#=*** constraint for each pair of variables
  - This constraint is materializable, and has a reified version
  - Examples:

```
| ?- domain([X,Y,Z], 1, 2),
     all_equal([X,Y,Z]),
     labeling([], [X, Y, Z]).
X = 1, Y = 1, Z = 1 ? ;
X = 2, Y = 2, Z = 2 ? ;
no
```

```
| ?- domain([X,Y,Z], 1, 2),
     all_equal([X, Y, Z]) #<=> B ,
     X #< Y,
     labeling([], [X, Y ,Z]).
X = 1, Y = 2, Z = 1, B = 0 ? ;
X = 1, Y = 2, Z = 2, B = 0 ? ;
no
```

All Different / All Distinct

- ***all_different(+Variables)*** / ***all_different(+Variables, +Options)***
- ***all_distinct(+Variables)*** / ***all_distinct(+Variables, +Options)***
  - True when all values in the ***Variables*** list are distinct
    - Equivalent to a **#\=** constraint for each pair of variables
  - ***Variables*** is a list of integers and/or domain variables
  - ***Options*** is a list with zero or more options (see documentation), controlling propagation, or adding additional side constraints
  - Examples:

```
| ?- domain([X,Y,Z], 1, 2),
     all_different([X,Y,Z]).
X in 1..2, Y in 1..2, Z in 1..2

| ?- domain([X,Y,Z], 1, 2),
     all_distinct([X,Y,Z]).
no
```

```
| ?- domain([X,Y,Z], 1, 3),
     all_different([X, Y, Z]),
     X #< Y,
     labeling([], [X]).
X = 1, Y in 2..3, Z in 2..3 ? ;
X = 2, Y = 3, Z = 1 ? ;
no
```

All Different / Distinct Except 0

- ***all_different_except_0(+Variables)***
- ***all_distinct_except_0(+Variables)***
  - ***Variables*** is a list of integers and/or domain variables
  - The constraint holds if the ***Variables*** list contains distinct values, with the exception of variables with the value 0
  - Examples:

```
| ?- L = [A,B,1,D], domain(L, 0, 2),
     all_distinct(L).
no

| ?-  L = [A,B,1,D], domain(L, 0, 2),
        all_distinct_except_0(L).
A in {0} \/ {2},
B in {0} \/ {2},
D in {0} \/ {2}  ?
```

```
| ?- L = [A,B,C,D], domain(L, 0, 2),
     all_distinct_except_0(L),
     labeling([], L).
L = [0,0,0,0] ? ;
L = [0,0,0,1] ? ;
L = [0,0,0,2] ? ;
L = [0,0,1,0] ? ;
L = [0,0,1,2] ? ;
L = [0,0,2,0] ? ;
L = [0,0,2,1] ? ;
L = [0,1,0,0] ? ;
...
```

Symmetric All Different / Distinct

- ***symmetric_all_different (+Variables)***
- ***symmetric_all_distinct (+Variables)***
  - ***Variables*** is a list of integers and/or domain variables
  - The constraint holds if all variables in the ***Variables*** list have distinct values, and for all variables Xi=j iff Xj=i
  - Examples:

```
| ?- L = [A,B,C,D],
     symmetric_all_distinct(L),
     A #= 3.
A = 3,
C = 1,
B in {2} \/ {4},
D in {2} \/ {4} ?
```

```
| ?- L = [A,B,C],
     symmetric_all_distinct(L),
     labeling([], L).
L = [1,2,3] ? ;
L = [1,3,2] ? ;
L = [2,1,3] ? ;
L = [3,2,1] ? ;
no
```

Assignment

- ***assignment(+Xs, +Ys)***
- ***assignment(+Xs, +Ys, +Options)***
  - ***Xs*** = [X1,...,Xn] and ***Ys*** = [Y1,...,Yn] are lists of length n of integers and/or domain variables
  - True if all ***Xi***, ***Yi*** are in [1,n], are unique within their list and ***Xi=j*** iff ***Yj=i*** (the lists are dual)
  - ***Options*** is a list that may contain, among others (see documentation), the options:
    - *circuit(Bool), subcircuit(Bool)*: if *true*, then *circuit(Xs, Ys)* / *subcircuit(Xs, Ys)* must hold
  - Examples:

```
| ?- assignment([4,1,5,2,3], Ys).
Ys = [2,4,5,1,3]
```

```
| ?- length(Xs, 3), domain(Xs, 1, 3),
     assignment(Xs, Ys), labeling([], Xs).
Xs = [1,2,3], Ys = [1,2,3] ? ;
Xs = [1,3,2], Ys = [1,3,2] ? ;
Xs = [2,1,3], Ys = [2,1,3] ? ;
Xs = [2,3,1], Ys = [3,1,2] ? ;
Xs = [3,1,2], Ys = [2,3,1] ? ;
Xs = [3,2,1], Ys = [3,2,1] ? ;
no
```

Sorting

- ***sorting(+Xs, +Ps, +Ys)***
  - Captures the relation between a list of values, a list of values ordered increasingly and the positions of the values in the original list
  - ***Xs***, ***Ps*** and ***Ys*** are lists of equal length n of integers and/or domain variables
  - The constraint holds if:
    - ***Ys*** is ordered increasingly; ***Ps*** is a permutation of [1..n]; For each i in [1..n], Xs[i] = Ys[Ps[i]]
  - Examples:

```
| ?- length(Ys, 5), length(Ps, 5),
     sorting([2,7,9,1,3], Ps, Ys).
Ys = [1,2,3,7,9],
Ps = [2,4,5,1,3] ?
```

```
| ?- length(Ys, 5), length(Ps, 5),
     sorting([2,7,3,1,3], Ps, Ys).
Ys = [1,2,3,3,7],
Ps = [2,5,_A,1,_B],
_A in 3..4, _B in 3..4 ?
```

Keysorting

- ***keysorting(+Xs, +Ys)***
- ***keysorting(+Xs, +Ys, +Options)***
  - Generalization of ***sorting/3*** but ordering tuples of variables
  - Tuples are separated into key and value, being ordered only by the key (maintains the order of tuples with the same key)
  - ***Xs*** and ***Ys*** are lists of the same size n of tuples of variables; all tuples (lists of variables) have the same size m
  - ***Options*** is a list of options:
    - ***keys(Keys)*** - ***Keys*** is the size of the key (positive integer; the default value is 1)
    - ***permutation(Ps)*** - Ps is a list of variables (permutation of [1..n], such that for each i in [1..n], j in [1..m] : Ys[i,j] = Xs[Ps[i],j].)
  - Example:

```
ln2(X):-
    length(X,2).
```

```
| ?- _Lst = [[1,5],[6,5],[4,3],[7,9],[4,5],[7,8],[3,3]],
     length(_Lst, _Len), length(Srt, _Len), maplist(ln2, Srt),
     length(P, _Len), keysorting(_Lst, Srt, [permutation(P)]).
Srt = [[1,5],[3,3],[4,3],[4,5],[6,5],[7,9],[7,8]]
P = [1,7,3,5,2,4,6] ? ;
no
```

Lex Chain

- ***lex_chain(+Vectors)***
- ***lex_chain(+Vectors, +Options)***
  - ***Vectors*** is a list of vectors (lists) of integers and/or domain variables
  - The constraint holds if ***Vectors*** is in an ascending lexicographic order (actually, non-decreasing by default)
  - ***Options*** is a list of options:
    - ***op(Op)*** - ***Op*** is ***#=<*** (default) or ***#<*** (strictly ascending)
    - ***increasing*** – internal lists ordered in a strictly ascending manner
    - ***among(Least, Most, Values)*** – between ***Least*** and ***Most*** values of each ***Vector*** belong to the ***Values*** list
  - Example:

```
| ?- domain([A,B,C], 1, 2),
     lex_chain([ [A,B,C], [B,C,A], [C,B,A] ]),
     labeling([], [A,B,C]).
A = 1,B = 1,C = 1 ? ;      A = 1,B = 1,C = 2 ? ;
A = 1,B = 2,C = 2 ? ;      A = 2,B = 2,C = 2 ? ;
no
```

Bool And / Or / Xor

- ***bool_and(+List, +Lit)***
- ***bool_or(+List, +Lit)***
- ***bool_xor(+List, +Lit)***
  - ***List*** is a list of literals, and ***Lit*** is a literal (here, a literal is a Boolean variable or its negation)
  - The constraint holds if the conjunction / disjunction / exclusive or of the values in ***List*** equals ***Lit***
  - It is usually more efficient than using the corresponding propositional constraints
  - Examples:

```
| ?- A in 1..2, B in 1..2,
     A #> 1 #<=> S, B #> 1 #<=> T,
     bool_xor([S, T], 1),
     labeling([], [A,B]).
A = 1, B = 2, S = 0, T = 1 ? ;
A = 2, B = 1, S = 1, T = 0 ? ;
no
```

```
| ?- A in 1..2, B in 1..2,
     A #> 1 #<=> S, B #> 1 #<=> T,
     bool_and([S, T], T),
     labeling([], [A,B]).
A = 1, B = 1, S = 0, T = 0 ? ;
A = 2, B = 1, S = 1, T = 0 ? ;
A = 2, B = 2, S = 1, T = 1 ? ;
no
```

Bool Channel

- ***bool_channel(+List, ?Y, +RelOp, +Offset)***
  - ***List*** is a list of literals, ***Y*** is a domain variable, ***RelOp*** a relational operator, and ***Offset*** an integer
  - The constraint holds if Li #<=> (Y RelOp i+Offset) for all Li in List
  - It is usually more efficient than using several reifications
  - Examples:

```
| ?- length(L, 5),
     bool_channel(L, 3, #=, 1).
L = [0,0,1,0,0] ? ;
no
| ?- bool_channel([0,0,0,1,0], X, #=, 1).
X = 4 ? ;
no
```

```
| ?- length(L, 5),
     bool_channel(L, 3, #>=, 1).
L = [1,1,1,0,0] ? ;
no
```

Cumulative

- ***cumulative(+Tasks)***
- ***cumulative(+Tasks, +Options)***
  - Constrains n task to be scheduled in such a way that the instantaneous resource consumption never exceeds a given limit at any time
  - ***Tasks*** is a list of n terms in the form ***task(Oi, Di, Ei, Hi, Ti)***
    - ***Oi*** = *start time*, ***Di*** = duration (non-negative), ***Ei*** = *end time*, ***Hi*** = resource consumption (non-negative), ***Ti*** = task identified (all fields can be integers or domain variables)
  - The constraint holds if, for all task i, *Oi+Di=Ei* and at any given time H1+H2+…+Hn =< L (resource limit, 1 by default)
    - ***Hi*** is only accounted for during the instants between ***Oi*** and ***Ei***; otherwise it's 0
  - ***Options*** is a list of options:
    - ***limit(L)***: L is the resource limit to use
    - ***precedences(Ps)***: precedence between tasks; ***Ps*** is a list of terms in the form ***Ti-Tj #= Dij***, with *Oi-Oj = Dij*
    - ***global(Boolean)***: if *true*, uses a more costly algorithm to obtain better interval pruning

Cumulative

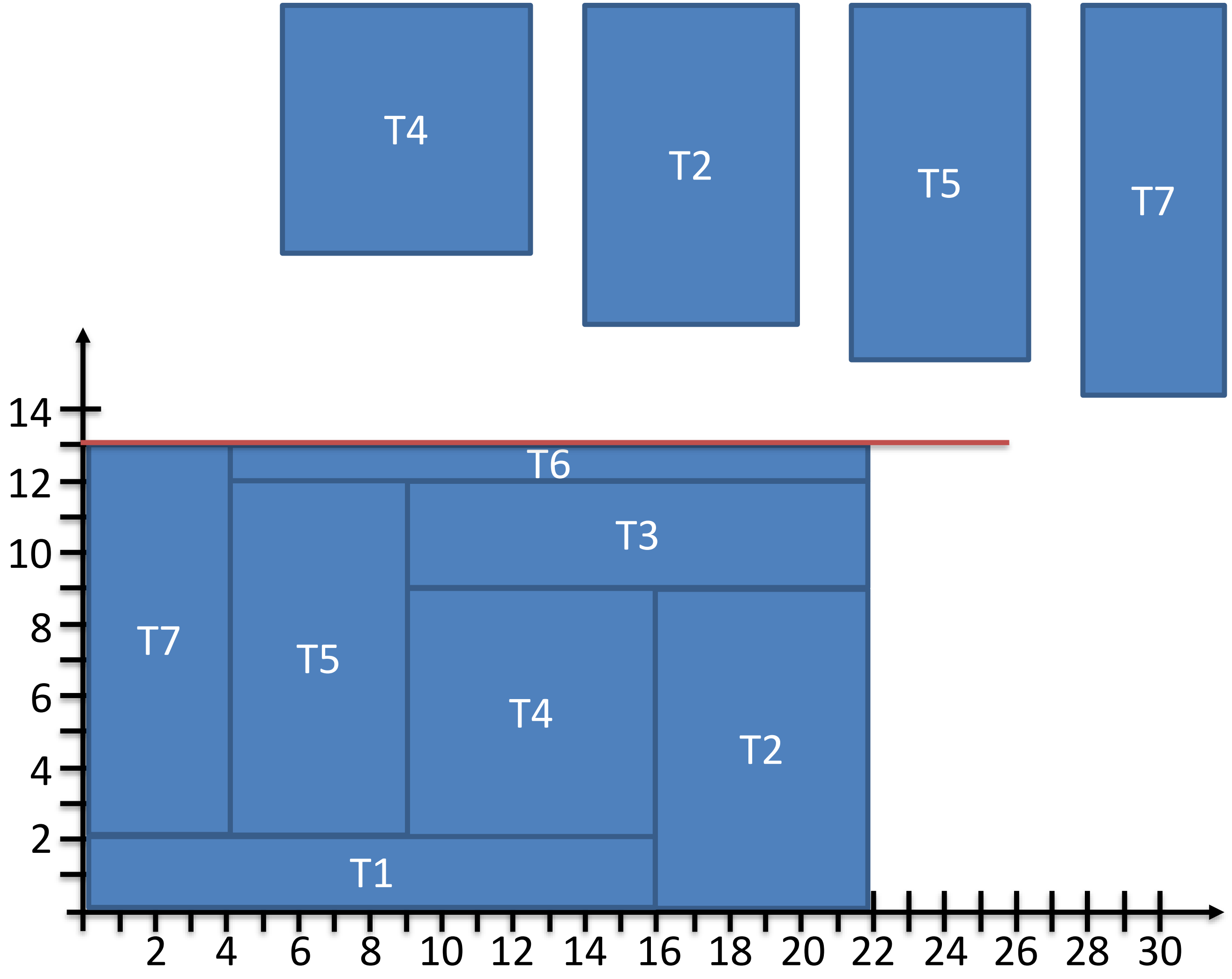
- Example:
  - Task scheduling:

Task	Duration	Resources
T1	16	2
T2	6	9
T3	13	3
T4	7	7
T5	5	10
T6	18	1
T7	4	11

- Resource limit = 13

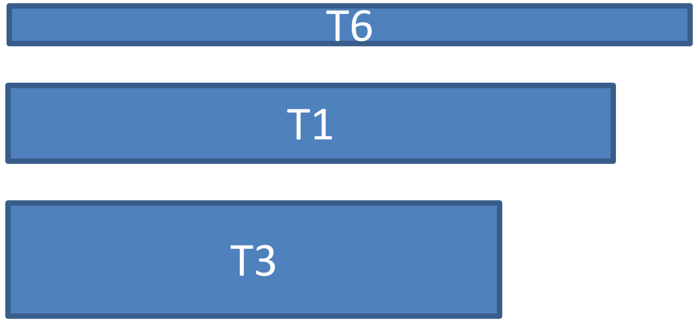
```
schedule(Ss, End) :-
    Ss = [S1,S2,S3,S4,S5,S6,S7],
    Es = [E1,E2,E3,E4,E5,E6,E7],
    Tasks = [
        task(S1, 16, E1,  2, 1),
        task(S2,  6, E2,  9, 2),
        task(S3, 13, E3,  3, 3),
        task(S4,  7, E4,  7, 4),
        task(S5,  5, E5, 10, 5),
        task(S6, 18, E6,  1, 6),
        task(S7,  4, E7, 11, 7) ],
    domain(Ss, 1, 30),
    maximum(End, Es),
    cumulative(Tasks, [limit(13)]),
    labeling([minimize(End)], Ss).
```

Cumulative



Task	Duration	Resources
T1	16	2
T2	6	9
T3	13	3
T4	7	7
T5	5	10
T6	18	1
T7	4	11

Resource limit = 13



Cumulatives

- ***cumulatives(+Tasks, +Machines)***
- ***cumulatives(+Tasks, +Machines, +Options)***
  - Constrains ***n*** tasks to be scheduled in ***m*** machines, where each machine has a (minimum or maximum) resource limit
  - ***Tasks*** is a list of terms in the form ***task(Oi, Di, Ei, Hi, Mi)***
    - ***Oi*** = *start time*, ***Di*** = duration (non-negative), ***Ei*** = *end time*, ***Hi*** = resource consumption (if positive) or resource production (if negative), ***Mi*** = machine identifier (all fields can be integers or domain variables)
  - ***Machines*** is a list of terms in the form ***machine(Mj, Lj)***
    - ***Mj*** = machine identifier (unique integer), ***Lj*** = machine resource limit (integer or domain variable with defined limits)
  - The constraint holds if for all tasks ***Oi+Di=Ei*** and in all machines and at all times ***H1m+H2m+…+Hnm >= Lm*** (if *lower bound*), or ***H1m+H2m+…+Hnm =< Lm*** (if *upper bound*)
  - ***Options*** is a list of options:
    - ***bound(B)*** – type of limit: ***lower*** (default) or ***upper***
    - ***prune(P)*** - ***all*** (default value) or ***next***: points to the pruning level to use
    - ***generalization(Boolean)***, ***task_intervals(Boolean)*** - if ***true*** some extra processing is performed

Cumulatives

- Example:
  - Task scheduling:

Task	Duration	Resources	Machine
T1	16	2	1
T2	6	9	2
T3	13	3	1
T4	7	7	2
T5	5	10	1
T6	18	1	2
T7	4	11	1

    - Resource limit for M1 = 12
    - Resource limit for M2 = 10

```
schedule(Ss, End) :-
    Ss = [S1,S2,S3,S4,S5,S6,S7],
    Es = [E1,E2,E3,E4,E5,E6,E7],
    Tasks = [
        task(S1, 16, E1,  2,  1),
        task(S2,  6, E2,  9,  2),
        task(S3, 13, E3,  3,  1),
        task(S4,  7, E4,  7,  2),
        task(S5,  5, E5, 10,  1),
        task(S6, 18, E6,  1,  2),
        task(S7,  4, E7, 11,  1) ],
    Machines = [machine(1,12),
        machine(2,10)],
    domain(Ss, 1, 30),
    maximum(End, Es),
    cumulatives(Tasks, Machines,
        [bound(upper)]),
    labeling([minimize(End)], Ss).
```

Multi_Cumulative

- ***multi_cumulative(+Tasks, +Capacities)***
- ***multi_cumulative(+Tasks, +Capacities, +Options)***
  - Generalization of the ***cumulative*** constraint, allowing tasks to consume multiple resources simultaneously; these can be of two types:
    - *cumulative* – resources as used in the ***cumulative*** / ***cumulatives*** constraints
    - *colored* – each task specifies a color (coded as an integer); the number of colors in use at each moment must not exceed a given limit; the color 0 means that the task does not use any color
  - ***Tasks*** is a list of terms in the form ***task(Oi, Di, Ei, Hsi, Ti)***
    - ***Oi*** = *start time*, ***Di*** = duration (non-negative), ***Ei*** = *end time*, ***Hsi*** = list of resource consumption/used color, ***Ti*** = task id
    - ***Oi*** and ***Ei*** are domain variables; the remaining fields must be integers
  - ***Capacities*** is a list of terms in the format ***cumulative(Limit)*** or ***colored(Limit)***
    - The size of the ***Capacities*** list must be equal to the size of all ***Hsi*** lists
  - The constraint holds if no resource exceeds its limit at any given time
  - ***Options*** is a list of options:
    - ***greedy(Flag)***: ***Flag*** is a variable with domain 0..1 denoting whether the greedy mode should be used
    - ***precedences(Ps)***: task precedence; ***Ps*** is a list of terms in the form ***Ti-Tj*** (***Ti*** and ***Tj*** are task identifiers) denoting that ***Ti*** must finish before ***Tj*** starts

Bin Packing

- ***bin_packing(+Items, +Bins)***
  - Assigns ‘items’ of certain sizes to ‘containers’ with given capacities
  - ***Items*** is a list of terms in the format ***item(Bin, Size)***
    - ***Bin*** is the container to which the item is assigned (integer or domain variable); ***Size*** is the item size (integer >=0)
  - ***Bins*** is a list of terms in the format ***bin(ID, Cap)***
    - ***ID*** is the identifier of each container (integer, all distinct); ***Cap*** is the container capacity (integer or domain variable)
  - The constraint holds if all items are assigned to an existing container and the sum of the sizes of all items assigned to each container equals its capacity
    - It may be required to use either ‘ghost objects’ or bins with variable capacities

Bin Packing

- Example:
  - 6 objects, 3 compartments

Item	Size
A	5
B	6
C	3
D	7
E	9
F	4

Bin	Cap
1	9
2	14
3	11

```
place(Vars) :-
    Vars = [A, B, C, D, E, F],
    Items = [ item(A, 5), item(B, 6),
        item(C, 3), item(D, 7),
        item(E, 9), item(F, 4) ],
    Bins = [ bin(1, 9),
        bin(2, 14),
        bin(3, 11) ],
    bin_packing(Items, Bins),
    labeling([], Vars).
```

```
| ?- place(Vars).
Vars = [2,1,1,3,2,3] ? ;
Vars = [2,2,2,3,1,3] ? ;
Vars = [3,3,2,2,1,2] ? ;
no
```

Disjoint

- ***disjoint1(+Lines)***
- ***disjoint1(+Lines, +Options)***
  - Constrains a set of lines in such a way that they do not overlap
    - 1D view of space (all lines are aligned)
  - ***Lines*** is a list of terms in the format ***F(Sj,Dj)*** or ***F(Sj, Dj, Tj)***
    - ***Sj*** and ***Dj*** represent origin and size of line j (integer or domain variable); ***F*** is any functor
    - ***Tj*** is an optional atomic term (0 by default) denoting the type of line
  - Options is a list of options
    - ***global(Boolean)*** - if ***true*** a redundant algorithm is used to attain a more complete pruning
    - ***wrap(Min, Max)*** – space is seen as a circle, where values ***Min*** and ***Max*** (integers) coincide; this option forces values back to interval [Min, Max-1]
    - ***margin(T1, T2, D)*** – imposes a minimum distance ***D*** between the end of any line of type ***T1*** and the beginning of any line of type ***T2***; ***D*** must be a positive integer or ***sup***: all lines of type ***T2*** end before any line of type ***T1***

Disjoint

– Example:

```
place(Starts) :-
    Starts = [A, B, C],
    domain(Starts, 1, 10),
    Lines = [
        line(A, 5),
        line(B, 7),
        line(C, 3)
    ],
    A #< C,
    disjoint1(Lines),
    labeling([], Starts).
```

```
Starts = [1,9,6] ? ;
Starts = [1,10,6] ? ;
Starts = [1,10,7] ? ;
Starts = [2,10,7] ? ;
no
```

```
place(Starts) :-
    Starts = [A, B, C, D],
    domain(Starts, 1, 12),
    Lines = [
        line(A, 4, r),
        line(B, 2, g),
        line(C, 3, r),
        line(D, 2, g)
    ],
    A #< B,
    disjoint1(Lines, [margin(r, g, 3)]),
    labeling([], Starts).
```

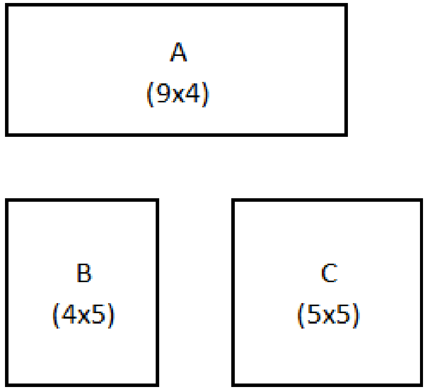
```
Starts = [1,8,12,10] ? ;
Starts = [1,10,12,8] ? ;
Starts = [3,10,12,1] ? ;
no
```

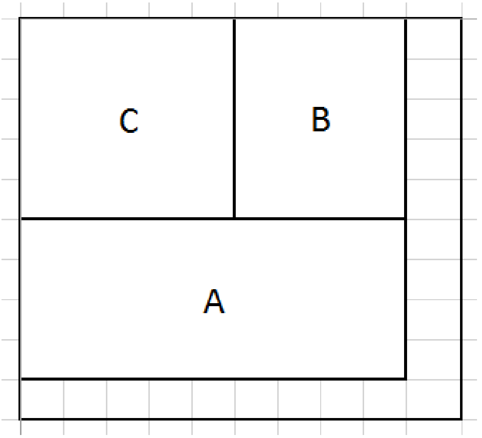
Disjoint

- ***disjoint2(+Rectangles)***
- ***disjoint2(+Rectangles, +Options)***
  - Constrains a set of rectangles in such a way that they do not overlap
  - ***Rectangles*** is a list of terms in the format ***F(Xj, Lj, Yj, Hj)*** or ***F(Xj, Lj, Yj, Hj, Tj)***
    - ***Xj*** and ***Yj*** represent the origin of rectangle j, while ***Lj*** and ***Hj*** represent its dimensions (integers or domain variables); ***F*** is any functor; ***Tj*** is an atomic term (0 by default) denoting the type of rectangle
  - Options is a list of options
    - ***wrap(Min1, Max1, Min2, Max2)*** - ***Min1*** and ***Max1*** refer to the X dimension, while ***Min2*** and ***Max2*** refer to the Y dimension; if all values are integers, the space is seen as a toroid; the values ***inf*** and ***<u>sup</u>*** can be used (for Min and Max in one of the dimensions) to obtain a cylindrical space
    - ***margin(T1, T2, D1, D2)*** – imposes a minimum distance ***D1*** in X and ***D2*** in Y between the end of any rectangle of type ***T1*** and the start of any rectangle of type ***T2***; ***D1*** and ***D2*** must be positive integers or ***sup***: all rectangles of type ***T2*** end before any rectangle of type ***T1*** in the relevant dimension

Disjoint

- Example:
  - Position three rectangles in a 10x10 grid






```
place(StartsX, StartsY) :-
    StartsX = [Ax, Bx, Cx],
    StartsY = [Ay, By, Cy],
    domain(StartsX, 1, 10),
    domain(StartsY, 1, 10),
    Rectangles = [
        rect(Ax, 9, Ay, 4),
        rect(Bx, 4, By, 5),
        rect(Cx, 5, Cy, 5) ],
    Ax + 9 #=< 10, Ay + 4 #=< 10,
    Bx + 4 #=< 10, By + 5 #=< 10,
    Cx + 5 #=< 10, Cy + 5 #=< 10,
    disjoint2(Rectangles),
    append(StartsX, StartsY, Vars),
    labeling([], Vars).
```

```
StartsX = [1,6,1],
StartsY = [6,1,1] ?
```

Diffn

- ***diffn(+Boxes)***
- ***diffn(+Boxes, +Options)***
  - Constrains the location in space of multidimensional boxes (***Boxes***) to not overlap
  - ***diffn/[1,2]*** should be used instead of ***disjoint1/[1,2]*** and ***disjoint2/[1,2]***
  - ***Boxes*** is a list of boxes, each represented by a list of terms in the format ***Origin-Length***
    - ***Origin*** and ***Length*** represent the origin and size of the box in each dimension
    - All boxes must have the same dimensionality (i.e., the lists must have the same size)
  - ***Options*** is a list of options
    - ***strict(Boolean)*** – if ***false***, the disjunction admits boxes with no length in some dimension(s); if ***true***, the disjunction is more strict

```
| ?- diffn([ [1-3, 1-3], [2-3, 4-3] ]).
yes
| ?- diffn([ [1-3, 1-3], [2-3, 2-3] ]).
no
```

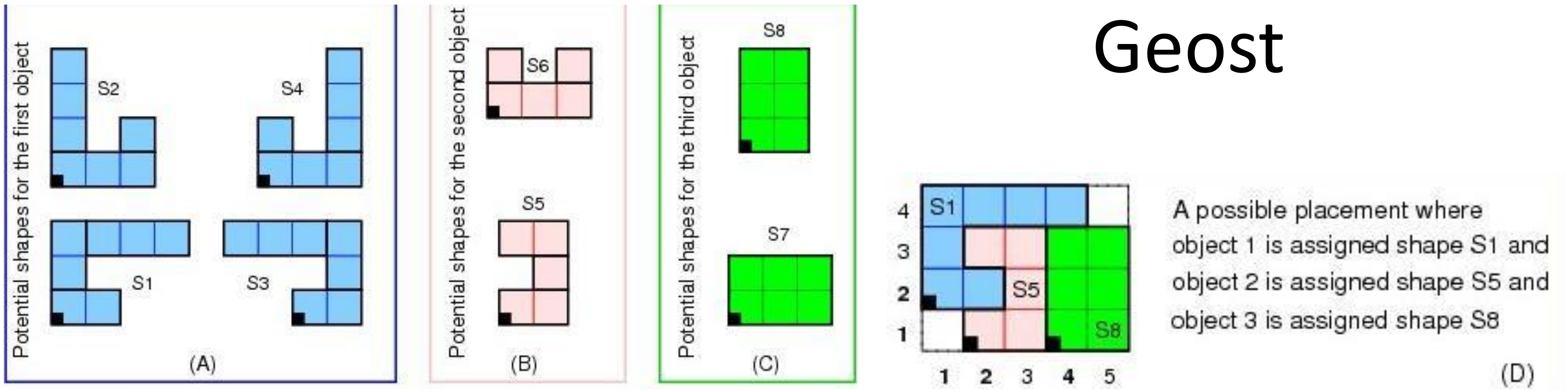
```
| ?- diffn([ [1-3, 1-0], [2-3, 0-3] ],
[strict(false)]).
yes
| ?- diffn([ [1-3, 1-0], [2-3, 0-3] ],
[strict(true)]).
no
```

Geost

- ***geost(+Objects, +Shapes)***
- ***geost(+Objects, +Shapes, +Options)***
- ***geost(+Objects, +Shapes, +Options, +Rules)***
  - Constrains the location in space of multidimensional objects (***Objects***) to not overlap, each object having a shape from a set of existing shapes (***Shapes***)
  - ***Objects*** is a list of terms in the format ***object(Oid, Sid, Origin)***
    - ***Oid*** identifies the object (unique integer); ***Sid*** identifies the shape of the object (integer or domain variable); ***Origin*** denotes the coordinates of the origin of the object (list of integers and/or domain variables)
  - ***Shapes*** is a list of terms in the format ***sbox(Sid, Offset, Size)***, representing *shifted boxes*
    - ***Sid*** is the identifier of the shape (integer); ***Offset*** is a list of integers of size n with the displacement in each dimension of the box, relative to the origin of the object; ***Size*** is a list of integers of size n with the size of the box in each dimension
    - Each shape is defined by the set of ***sbox/3*** terms with the same ***Sid***
  - ***Options*** is a list of options (see documentation)

Geost

- Example:




```
| ?- domain([X1,X2,X3,Y1,Y2,Y3],1,4),
     S1 in 1..4, S2 in 5..6, S3 in 7..8,
     geost( [ object(1,S1,[X1,Y1]), object(2,S2,[X2,Y2]), object(3,S3,[X3,Y3]) ],
       [ sbox(1,[0,0],[2,1]), sbox(1,[0,1],[1,2]), sbox(1,[1,2],[3,1]), % first object, shape S1
         sbox(2,[0,0],[3,1]), sbox(2,[0,1],[1,3]), sbox(2,[2,1],[1,1]), % first object, shape S2
         sbox(3,[0,0],[2,1]), sbox(3,[1,1],[1,2]), sbox(3,[2,2],[3,1]), % first object, shape S3
         sbox(4,[0,0],[3,1]), sbox(4,[0,1],[1,1]), sbox(4,[2,1],[1,3]), % first object, shape S4
         sbox(5,[0,0],[2,1]), sbox(5,[1,1],[1,1]), sbox(5,[0,2],[2,1]), % second object, shape S5
         sbox(6,[0,0],[3,1]), sbox(6,[0,1],[1,1]), sbox(6,[2,1],[1,1]), % second object, shape S6
         sbox(7,[0,0],[3,2]),    % third object, shape S7
         sbox(8,[0,0],[2,3])     % third object, shape S8
       ]),
     labeling([],[X1,X2,X3,Y1,Y2,Y3]).
```

Circuit

- ***circuit(+Succ)***
- ***circuit(+Succ, +Pred)***
  - ***Succ*** is a list of length n of integers and/or domain variables
  - The i^{th} element of ***Succ*** (***Pred***) is the successor (predecessor) of i in the graph
  - The constraint holds if the values form a Hamiltonian circuit
    - Nodes are numbered from 1 to n, the circuit starts in node 1, visits each node and returns to the origin
  - Examples:

```
| ?- length(L, 5), domain(L, 1, 5),
     circuit(L).
L = [ _A,_B,_C,_D,_E ],
_A in 2..5,
_B in {1}\/(3..5),
_C in (1..2)\/(4..5),
_D in (1..3)\/{5},
_E in 1..4 ?
yes
```

```
| ?- length(L, 5),
     domain(L, 1, 5),
     circuit(L),
     labeling([], L).
L = [2,3,4,5,1] ? ;
L = [2,3,5,1,4] ? ;
…
```

Subcircuit

- ***subcircuit(+Succ)***
- ***subcircuit(+Succ, +Pred)***
  - ***Succ*** is a list of length n of integers and/or domain variables
  - The i^th^ element of ***Succ*** (***Pred***) is the successor (predecessor) of i in the graph; or i if the element is not included in the sub-circuit
  - The constraint holds if the values included form at most one Hamiltonian circuit
  - Examples:

```
| ?- length(L,5),
     domain(L,1,5),
     circuit(L).
L = [ _A,_B,_C,_D,_E ],
_A in 2..5,
_B in {1}\/(3..5),
_C in (1..2)\/(4..5),
_D in (1..3)\/{5},
_E in 1..4 ?
```

```
| ?- length(L,5),
     domain(L,1,5),
     subcircuit(L).
L = [ _A,_B,_C,_D,_E ],
_A in 1..5,
_B in 1..5,
_C in 1..5,
_D in 1..5,
_E in 1..5 ?
```

Value Precede Chain

- ***value_precede_chain(+Values, +Vars)***
- ***value_precede_chain(+Values, +Vars, +Options)***
  - Provides a way of removing value symmetries
  - ***Values*** is a list of integers and ***Vars*** is a list of integers and/or domain variables
  - The constraint holds if for each pair of adjacent values ***X***, ***Y*** in ***Values***, ***Y*** does not exist in ***Vars***, or, if ***Y*** exists in ***Vars***, ***X*** appears before ***Y***
  - ***Options*** is a list of options:
    - ***global(Bool)***: if ***false*** (default value) a decomposition of the constraint into ***automaton/3*** is performed. If ***true***, a custom algorithm is used. Both maintain domain consistency, but the relative performance may vary
  - Examples:

```
| ?- length(L, 3),
     domain(L, 1, 2),
     value_precede_chain([3,2,1], L).
no
```

```
| ?- length(L, 3),
     domain(L, 1, 3),
     value_precede_chain([3,4,2,1], L).
L = [3,3,3] ? ;
no
```

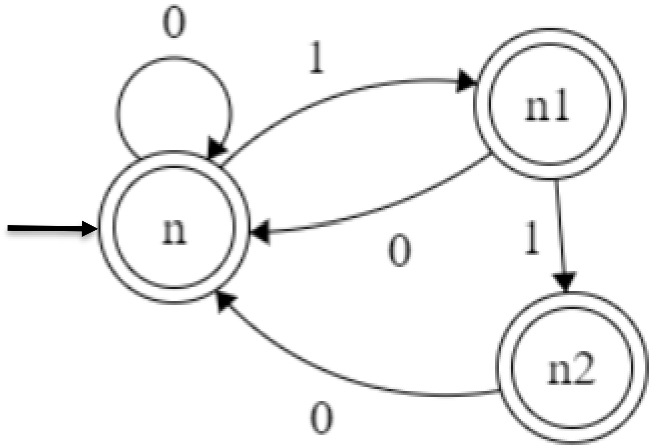
Sequence Precede Chain

- ***seq_precede_chain(+Vars)***
- ***seq_precede_chain(+Vars, +Options)***
  - Similar to the previous constraint, assuming ***Values*** = [1, 2, 3, ...]

Automaton

- ***automaton(Signature, SourcesSinks, Arcs)***
- ***automaton(Sequence, Template, Signature, SourcesSinks, Arcs, Counters, Initial, Final)***
- ***automaton(Sequence, Template, Signature, SourcesSinks, Arcs, Counters, Initial, Final, Options)***
  - General method of defining any constraint involving sequences that can be verified by a finite automaton, deterministic or not, extended with possible counting operations in the edges
  - If counters are not used, it maintains domain consistency
  - ***Signature*** is a sequence of integers and/or domain variables, based on which the transitions in the automaton will be performed
  - ***SourcesSinks*** is a list of elements in the form ***source(node)*** or ***sink(node)***, identifying the initial and acceptance nodes of the automaton, respectively
  - ***Arcs*** is a list of elements in the form ***arc(node, integer, node)*** or ***arc(node, integer, node, exprs)***, identifying the possible transitions between nodes and possible operations over variables in ***Counters***
  - ***Counters***, ***Initial*** and ***Final*** are lists of equal size identifying counter variables, their initial values (usually instantiated) and final values (usually non-instantiated), respectively
  - ***Options*** is a list of options (see documentation)

Automaton



- Example:

```
at_most_two_consecutive_ones(Vars) :-
    automaton(Vars,
        [source(n), sink(n), sink(n1), sink(n2)],
        [arc(n, 0, n),    arc(n, 1, n1),
         arc(n1, 1, n2),  arc(n1, 0, n),
         %arc(n2, 1, false),
         arc(n2, 0, n) ]).
```

```
| ?- at_most_two_consecutive_ones([0,0,0,1,1,1]).
no
| ?- at_most_two_consecutive_ones([0,1,1,0,1,1]).
yes
| ?- at_most_two_consecutive_ones([0,1,1,0,1,0]).
yes
```

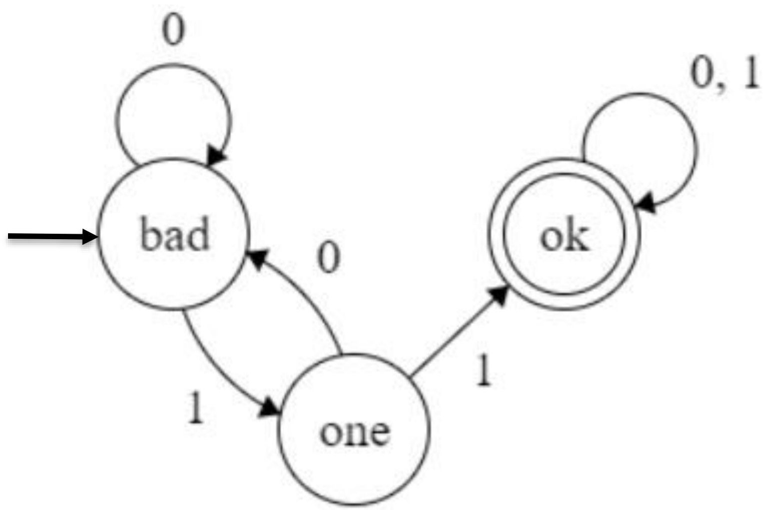
```
| ?- length(L,3),
     at_most_two_consecutive_ones(L).
L = [_A,_B,_C],
_A in 0..1, _B in 0..1, _C in 0..1

| ?- length(L,3),
     at_most_two_consecutive_ones(L),
     L = [1|_], labeling([], L).
L = [1,0,0] ? ;
L = [1,0,1] ? ;
L = [1,1,0] ? ;
No
```

Automaton

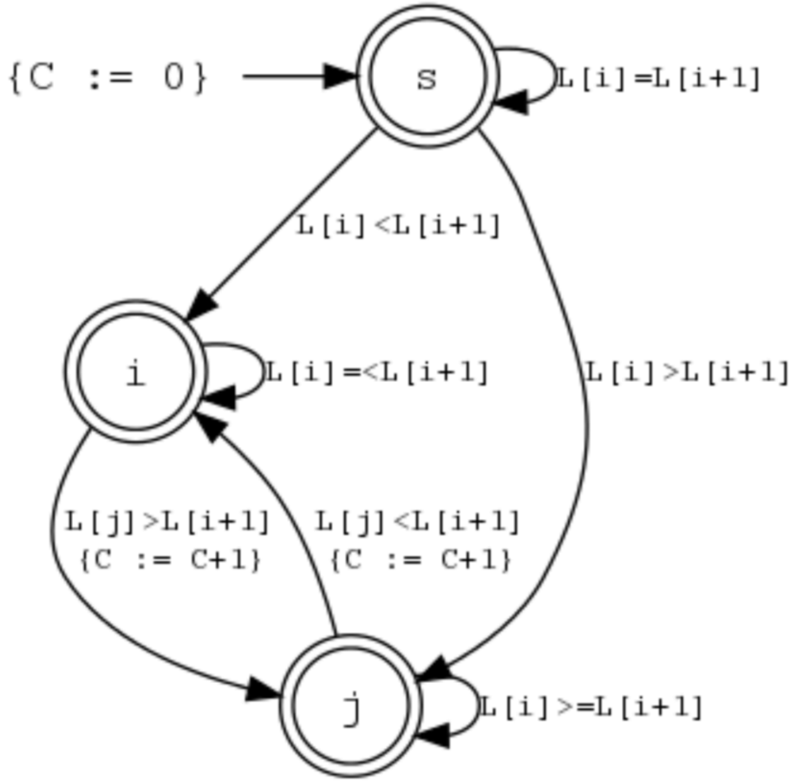
- Example:

```
at_least_two_consecutive_ones(Vars, N) :-
    length(Vars, N),
    %domain(Vars, 0, 1),
    automaton(Vars,
        [source(bad), sink(ok)],
        [arc(bad, 0, bad),  arc(bad, 1, one),
         arc(one, 0, bad),  arc(one, 1, ok),
         arc(ok, 0, ok),    arc(ok, 1, ok)]),
    labeling([], Vars).
```




```
| ?- at_least_two_consecutive_ones(L, 3).
L = [0,1,1] ? ;
L = [1,1,0] ? ;
L = [1,1,1] ? ;
no
```

Automaton




```
inflexion(N, Vars) :-
    inflexion_signature(Vars, Sign),
    automaton(Sign, _, Sign,
        [source(s), sink(i), sink(j), sink(s)],
        [arc(s,1,s), arc(s,2,i), arc(s,0,j),
         arc(i,1,i), arc(i,2,i), arc(i,0,j,[C+1]),
         arc(j,1,j), arc(j,0,j), arc(j,2,i,[C+1])],
        [C], [0], [N]).

inflexion_signature([], []).
inflexion_signature([_], []) :- !.
inflexion_signature([X, Y |Ys], [S|Ss]) :-
   S in 0..2,
   X #> Y #<=> S #= 0,
   X #= Y #<=> S #= 1,
   X #< Y #<=> S #= 2,
   inflexion_signature([Y|Ys], Ss).
```

```
| ?- inflexion(N, [1,1,4,8,8,2,7,1]).
N = 3

| ?- length(L,4), domain(L,0,1),
     inflexion(2,L), labeling([],L).
L = [0,1,0,1] ? ;
L = [1,0,1,0] ? ;
no
```

Regular

- ***regular(Signature, RegExpr)***
  - Alternative (and more compact) manner of defining an automaton, by using a regular expression
  - ***Signature*** is a sequence of integers and/or domain variables to validate against the regular expression
  - ***RegExpr*** is a ground Prolog term representing the regular expression (see documentation)
  - The constraint holds if ***Signature*** matches ***RegExpr***

```
| ?- regular([3,2,5],
+{1,2}+(+{3,4})+(+{5})).
no

| ?- regular([1,3,5],
+{1,2}+(+{3,4})+(+{5})).
yes
```

```
| ?- length(L, 3), domain(L, 1, 10),
     regular(L, +{1,2}+(+{3,4})+(+{5})),
     labeling([], L).
L = [1,3,5] ? ;
L = [1,4,5] ? ;
L = [2,3,5] ? ;
L = [2,4,5] ? ;
no
```

Element

- ***element(?X, +List, ?Y)***
- ***element(+List, ?Y)***
  - ***X*** and ***Y*** are integers and/or domain variables; ***List*** is a list of integers and/or domain variables
  - True if the ***X***th element of ***List*** is ***Y*** / true if ***Y*** exists in ***List***
  - Operationally, the domains of ***X*** and ***Y*** are constrained in such a way that, for each element in the domain of ***X***, there is a compatible element in the domain of ***Y***, and vice-versa
  - Maintains domain consistency on ***X*** and interval consistency in ***List*** and ***Y***
  - Corresponds to ***nth1/3*** from *library(lists)*.
  - Examples:

```
| ?- element(X, [10,20,30], Y),
     labeling([], [Y]).
X = 1, Y = 10 ? ;
X = 2, Y = 20 ? ;
X = 3, Y = 30 ? ;
no
```

```
| ?- L=[A, B, C],
     domain(L, 1, 5),
     element(2, L, 4).
B = 4,
L = [A,4,C],
A in 1..5,
C in 1..5 ?
```

Relation

(*deprecated*, see *table*)

- ***relation(?X, +MapList, ?Y)***
  - ***X*** and ***Y*** are integers or domain variables and ***MapList*** is a list of pairs *Integer-ConstantRange*, where each integer key occurs only once
  - The constraint holds if ***MapList*** contains a pair ***X-R*** and ***Y*** is in the interval stated by ***R***
  - Examples:

```
| ?- domain([Y], 1, 3),
     relation(X, [1-{3,4,5}, 2-{1,2}], Y),
     labeling([], [X]).
X = 1, Y = 3 ? ;
X = 2, Y in 1..2 ? ;
no
```

```
| ?- domain([Y], 1, 3),
     relation(X, [1-{3,4,5},
          2-{1,2,3}], Y),
     labeling([], [Y]).
Y = 1, X = 2 ? ;
Y = 2, X = 2 ? ;
Y = 3, X in 1..2 ? ;
no
```

Table

- ***table(+Tuples, +Extension)***
- ***table(+Tuples, +Extension, +Options)***
  - Defines an n-ary constraint by extension
  - ***Tuples*** is a list of lists of integers and/or domain variables, each of length ***n***; ***Extension*** is a list of lists of integers, each of length ***n***; ***Options*** is a list of options that allow controlling the order of the variables used internally and the data structure and algorithm (see documentation)
  - The constraint holds if each *Tuple* in ***Tuples*** occurs in ***Extension***
  - Examples:

```
| ?- table([[A,B]],[[1,1],[1,2],[2,10],[2,20]]).
A in 1..2,
B in (1..2)\/{10}\/{20}

| ?- table([[A,B],[B,C]],[[1,1],[1,2],[2,10],[2,20]]).
A = 1,
B in 1..2,
C in (1..2)\/{10}\/{20}
```

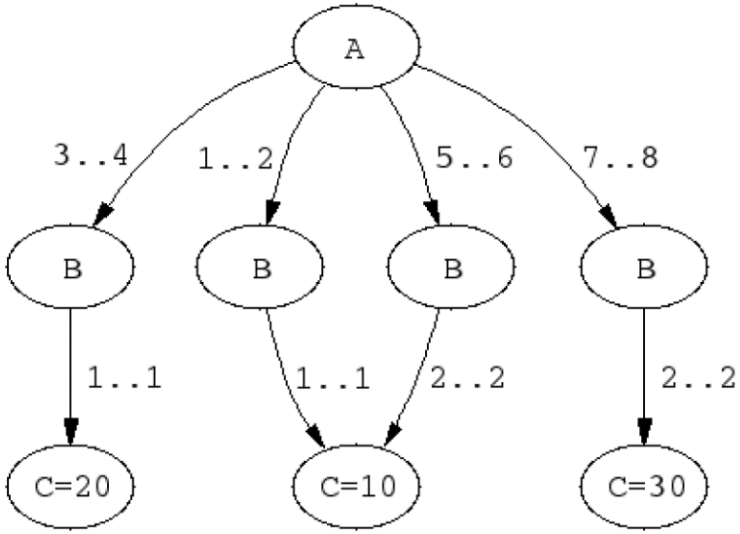
```
| ?- table([[A,B]], [[1,1], [1,2],
          [2,10], [2,20]]),
     labeling([], [A,B]).
A = 1, B = 1 ? ;
A = 1, B = 2 ? ;
A = 2, B = 10 ? ;
A = 2, B = 20 ? ;
no
```

Case

- ***case(+Template, +Tuples, +Dag)***
- ***case(+Template, +Tuples, +Dag, +Options)***
  - Codes an n-ary constraint, defined by extension and/or linear inequalities
  - Uses a DAG: the nodes correspond to variables, each arc is labeled by an admissible interval to the variable in the node of origin, or by linear inequalities
  - Variable order is fixed: each path from the root to a leaf must visit each variable once, in the order by which they appear in ***Template***
  - ***Template*** is an arbitrary non-ground term
  - ***Tuples*** is a list of terms in the same form as in ***Template*** (they should not share variables)
  - ***Dag*** is a list of terms in the form ***node(ID, X, Children)***, where ***X*** is a variable from template and ***ID*** an integer identifying the node; the first node in the list is the root
    - Internal node: ***Children*** is a lista of terms *(Min..Max)-ID2* (or *(Min..Max)-SideConstraints-ID2*), where *ID2* identifies a child node
    - Leaf node: ***Children*** is a list of terms *(Min..Max)* (or *(Min..Max)-SideConstraints*)

Case

– Example:




```
element(X, [1,1,1,1,2,2,2,2], Y),
element(X, [10,10,20,20,10,10,30,30], Z)
```

```
elts(X, Y, Z) :-
    case(f(A,B,C), [f(X,Y,Z)],
         [node(0, A,[(1..2)-1,(3..4)-2,(5..6)-3,(7..8)-4]),
          node(1, B,[(1..1)-5]),
          node(2, B,[(1..1)-6]),
          node(3, B,[(2..2)-5]),
          node(4, B,[(2..2)-7]),
          node(5, C,[(10..10)]),
          node(6, C,[(20..20)]),
          node(7, C,[(30..30)])]).
```

```
| ?- elts(X, Y, Z).
X in 1..8,
Y in 1..2,
Z in {10}\/{20}\/{30}

| ?- elts(X, Y, Z), Z #>= 15.
X in(3..4)\/(7..8),
Y in 1..2,
Z in {20}\/{30}

| ?- elts(X, Y, Z), Y = 1.
Y = 1,
X in 1..4,
Z in {10}\/{20}
```

CLP in SICStus Prolog

4. ENUMERATION PREDICATES

Search

- Usually constraint *solvers* for finite domains are not complete, i.e., they do not guarantee that the set of constraints has a solution
- Search (enumeration) is necessary to verify the satisfiability and obtain concrete solutions
- Predicates to perform search:
  - ***indomain(?X)***
    - X is an integer or domain variable
    - Assigns, via backtracking, admissible values to X, in increasing order
  - ***labeling(:Options, +Variables)***
  - ***solve(:Options, :Searches)***

Search

- ***labeling(:Options, +Variables)***
  - ***Options*** is a list of search options
  - ***Variables*** is a list of integers and/or domain variables
  - The predicate succeeds if it can find [at least] one attribution of values to the variables that satisfies all constraints, failing if there is no solution / if no solution is found within the time limits
  - Examples:

```
| ?- declareVariables(Vars),
     postConstraints(Vars),
     labeling([], Vars).
```

```
| ?- declareVariables(Vars),
     postConstraints(Vars),
     objectiveFunction(Vars, Profit),
     labeling( [maximize(Profit), ffc,
        bisect, time_out(5000,Flag)], Vars).
```

Search Options

- The ***Options*** argument of *labeling/2* (also used in *solve/2*) controls several parameters of the search
  - Variable ordering
  - Value selection
  - Value ordering
  - Solutions to find
  - Search time limit
  - Search scheme (useful in optimization problems):
    - *bab* (uses *branch-and-bound*; value by default), *restart*
  - Assumptions:
    - *assumptions(K): K* is unified with the number of choices made
  - Discrepancy:
    - *discrepancy(D)*: in the path to the solution there are at most *D* choice points in which there was backtracking

Variable Ordering

- How to select the next variable?
  - ***leftmost*** (default option): leftmost variable from the variable list
  - ***min***: variable with the smallest value in its domain (smallest lower bound)
  - ***max***: variable with the greatest value in its domain (greatest upper bound)
  - ***ff***: first-fail principle - variable with the smallest domain (fewer possible values)
  - ***anti_first_fail***: variable with the largest domain (more possible values)
  - ***occurrence***: variable with more suspended constraints
  - ***ffc***: variable with the smallest domain, breaking ties by choosing the one with more suspended constraints
  - ***max_regret***: variable with the largest difference between the first two values in its domain

Variable Ordering

- How to select the next variable? (continued)
  - ***impact***: variable that has been involved in the most failures
  - ***dom_w_deg***: variable with the largest (failure count / domain size)
  - ***variable(Sel)***:
    - ***Sel*** is a predicate used to select the next variable, with signature *Sel(Vars, Selected, Rest)*
    - Must succeed determinately, unifying *Selected* with the selected variable, and *Rest* with a list containing the remaining variables
    - Example:

```
...
labeling([variable(selRandom)], Vars).

% selects a variable randomly
selRandom(ListOfVars, Var, Rest):-
    random_select(Var, ListOfVars, Rest).    % from library(random)
```

Value Selection

- How to select the next value for the current variable?
  - ***step*** (default option): binary choice between X #= B and X #\= B, where B is the lower or upper *bound* of the domain of X
  - ***enum***: multiple choice for X corresponding to the values of its domain
  - ***bisect***: binary choice between X #=< M and X #> M, where M is the middle point of the domain of X (mean between the minimum and maximum values of the domain of X, rounded down)
  - ***median*** / ***middle***: binary choice between X #= M and X #\= M, where M is the median / average of the domain of X

Value Selection

- How to select the next value for the current variable?
  - ***value(Enum)***:
    - ***Enum*** is a predicate that must reduce the domain of X with signature *Enum(X, Rest, BB0, BB)*
      - *Rest* is the list of variables that need *labeling* with the exception of *X*
    - ***Enum*** must succeed in a non-determinate manner, providing other means of domain reduction by *backtracking*
      - Must call *first_bound(BB0, BB)* in its first solution and *later_bound(BB0, BB)* in alternate solutions
    - Example:

```
...
labeling( [ value(selRandom) ], Vars).

selRandom(Var, Rest, BB0, BB1):-          % selects value randomly
     fd_set(Var, Set), fdset_to_list(Set, List),
     random_member(Value, List),          % da library(random)
     ( first_bound(BB0, BB1), Var #= Value ;
       later_bound(BB0, BB1), Var #\= Value ).
```

Value Ordering

- In which order should the next value for the current variable be selected?
  - (not useful with the *value(Enum)* option)
  - ***up*** (default value): the domain is explored in ascending order
  - ***down***: the domain is explored in descending order

Solutions to Find

- These options indicate if the problem is a constraints satisfaction problem (any solution is valid) or an optimization problem (only the best solution matters):
  - ***satisfy*** (default value): all solutions are enumerated by backtracking
  - ***minimize(X)*** / ***maximize(X)***: the goal is to obtain a solution that minimizes / maximizes the domain variable ***X***
    - The *labeling* mechanism must constrain X to a value for every variable value attributions
    - It is useful to combine this option with *time_out/2*, *best* or *all*
- Options that only make sense with optimization problems:
  - ***best*** (default option): obtains the optimal solution, after proving optimality
  - ***all***: obtains, via *backtracking*, solutions that improve on the previous one

Search Time Limit

- The ***time_out(Time, Flag)*** flag defines a time limit for the search
  - ***Time*** is the maximum execution time (in milliseconds)
  - If the solver proves that there is no solution to the problem within ***Time*** ms, the predicate fails
  - If the time limit is reached, or the optimal solution is found, the predicate succeeds and ***Flag*** is unified with one of the following values:
    - ***optimality*** – the optimal solution has been found (the ***best*** option was selected) within the time limit; the variables are unified with the values corresponding to the best solution
    - ***success*** – at least one solution was found (but not proof of optimality) within the time limit; the variables are unified with the values corresponding to the best solution found until then
    - ***time_out*** – the time limit was reached, with no solution found; the variables remain uninstantiated

Search

- ***solve(:Options, :Searches)***
  - ***Options*** is a list of search options (similar to the ones used in *labeling/2*)
  - ***Searches*** is a list of one or more *labeling/2* or *indomain/1* goals
  - Used primarily in optimization problems, allowing the definition of different search heuristics for distinct [sets of] variables
  - Some options are global, while the majority are local
    - Global options in ***Options*** override the options in the individual *labeling/2* goals in ***Searches***
    - Local options in ***Options*** define default values for search options not appearing in the individual *labeling/2* goals in ***Searches***

Optimization

- The optimization predicates allow the search for an optimal solution (minimization / maximization of cost / profit):
  - **minimize(:Goal, ?X)** / **minimize(:Goal, ?X, +Options)**
  - **maximize(:Goal, ?X)** / **maximize(:Goal, ?X, +Options)**
    - Use a *branch-and-bound* algorithm to search for the assignment that minimizes/maximizes domain variable ***X***
    - ***Goal*** must be a goal that constrains ***X*** to a value, typically a *labeling/2* goal
    - The algorithm calls ***Goal*** repeatedly with an *upper* (*lower*) *bound* of ***X*** progressively more constrained until a proof of optimality is attained (which, sometimes, can take too long…)
    - ***Options*** is a list containing one of:
      - *best* (default option): returns the optimal solution after proof of optimality
      - *all*: enumerates improving solutions until proof of optimality

Examples

- Enumerate solutions with static variable ordering:

```
| ?- constraints(Variables),
     labeling( [], Variables ).
```

[] is the same as **[leftmost, step, up, satisfy]**

- Minimize a cost function, obtaining only the best solution, dynamic variable ordering using the first-fail principle, and dividing the domain by exploring the upper part of the domain first:

```
| ?- constraints(Variables, Cost),
     labeling([ff,bisect,down,minimize(Cost)], Variables).
```

Examples

- Minimize the cost, using two different search strategies for two variable subsets:
  - ```
    | ?- constraints(A, B, C, D, E, F),
         solve( [minimize(Cost)],
            [ labeling([ffc, bisect], [A, C, E]),
              labeling([max_regret,median], [B, D, F]) ]
         ).
    ```
    ```

CLP in SICStus Prolog

5. STATISTICS PREDICATES

Statistics Predicates

- Execution statistics specific to the *clp(fd)* solver:
  - ***fd_statistics(?Key, ?Value)***: for each possible key ***Key***, ***Value*** is unified with the current value of a counter:
    - ***resumptions***: number of times a constraint was resumed
    - ***entailments***: number of times a *(dis)entailment* was detected
    - ***prunings***: number of times a domain has been reduced
    - ***backtracks***: number of times a contradiction was found (domain depletion or constraint failing)
    - ***constraints***: number of created constraints
  - ***fd_statistics/0***: shows a summary of the statistics above (values since the last call to the predicate)

Statistics Predicates

- Other statistics relative to CPU time, memory use and others can be obtained with the predicates:
  - ***statistics(?Keyword, ?List))***: for each possible key ***Keyword***, ***List*** is unified with the current value of a counter. Examples:
    - ***runtime*** / ***total_runtime*** / ***walltime***: execution time (in ms) excluding memory management and system calls / total execution time / absolute time. The first element of the list refers to the time since the beginning of the session, while the second refers to the time since the last call to the *statistics* predicate.
    - ***memory_used***: used memory (in bytes)
    - Several other options are described in section 4.10.1.2 of SICStus' manual
  - ***statistics/0*** shows a summary of statistics related to execution time, memory, garbage collection, ...

Example

```
testStats(Vars):-
    declareVars(Vars),
    reset_timer,
    postConstraints(Vars),
    print_time('Posting Constraints: '),
    labeling([], Vars),
    print_time('Labeling Time: '),
    fd_statistics,
    statistics.
```

```
reset_timer:-
    statistics(total_runtime, _).

print_time(Msg):-
    statistics(total_runtime,[_,T]),
    TS is ((T//10)*10)/1000, nl,
    write(Msg),
    write(TS),
    write('s'), nl, nl.
```

CLP in SICStus Prolog

Q & A